E U R O P E A N   S O U T H E R N   O B S E R V A T O R Y

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

VERY LARGE TELESCOPE

VLT Software

CCS - LCU / Motor Control Module - Part I

Application Programmatic & Command Interface

User Manual

Doc.No. VLT-MAN-ESO-17210-0600

Issue 1.7

Date 02/10/98

Prepared P. Duhoux 02/10/98 ....

Name Date Signature

Approved A. Longinotti /10/98 ....

Name Date Signature

Released G. Raffi /10/98 ....

Name Date Signature

Change Record

Issue/Rev.	Date	Section/Page affected	Reason/Initiation/Document/Remarks
1.0	06/04/94	All	First issue
1.1	07/03/95	All	Update to MCM Version 1.2 - new: Command Interface SDL User Manual is now a separate document VLT-MAN-ESO-17210-0776
1.2	04/08/95	2, 3, 4 4	Upgrade to MCM Version 1.3 - JUL95 Complete database description in section 4 including SDL part. (Removed from VLT-MAN-ESO-17210-0776)
1.3	20/01/96	 2.3 2.4 3.1.23 3.1.39 and 3.1.40 4.6 4.8	Update for MCM Version 1.4 - DEC95 Motor registration Enhanced description New man-page motNameToVel() Upgraded man-pages motWaitInit/Move() Database description extended Upgrade to SDL part: MAC4 for STP/SSI
1.4	08/05/96	 2.4.5 2.4.13 3 4.2 4.8	Update for MCM Version 1.5 - JUN96 Section in indexed positions External encoder support Updated man-pages Added new classes Upgrade SDL part: -MAC4 for INC/SSI 4.1 & STP 2.0
1.5	16/10/96	 2.4.5 3	Update for MCM Version 1.6 - NOV96 Section in status Updated man-pages
1.6	17/11/97	 2.3 3.3 3.4 5.3	Update for MCM Version 1.7 - NOV97 Motor configuration/registration Updated man-pages (motInitDb/motMotors) Include files Motor instanciation/installation

Change Record

Issue/Rev.	Date	Section/Page affected	Reason/Initiation/Document/Remarks
1.7	02/10/98		Update for MCM Version 2.7 - OCT98
		1.3	Maccon User Manual in English also
		2.1	Module versions updated
		2.4.3	Documented action **`motACTION_DELAY`**
		2.4.5	Documented **`motOFF_REFERENCE_SWITCH`** motion
		3	Updated man-pages
		3.1.19	New motInitSampling(3) man-page
		3.4	Updated include files
		4.5	General Status section motSTATUS.class
		4.6.1	Updated attribute **`SERVER:CONF.motor`**
		4.6.2	Updated attribute **`SERVER:INIT.actions`**
		5	Updated versions to OCT98

1 INTRODUCTION

The software described in this manual is intended to be used in the ESO VLT project by ESO and authorized external contractors only.

While every precaution has been taken in the development of the software and in the preparation of this documentation, ESO assumes no responsibility for errors or omissions, or for damage resulting from the use of the software or of the information contained herein.

1.1 Purpose

This document is the first part of the User Manual of the **Motor Control Module** software. It is dedicated to the description of the interface to higher level applications implemented on LCU or on WS. The interface is realized by two SW modules:

- **mot** is the Application Programmatic Interface for LCU applications (functions are programmatically invoked).
- **motci** is the Application Command Interface for LCU and WS applications. This module consists in a LCU process (built using the LCC Command Interpreter) implementing in form of commands most of the API functions.

The present document provides all information necessary to implement applications dealing with motion control.

The author assumes that the reader has a good knowledge of UNIX, C-language, the VxWorks operating system and is familiar with the VxWorks development environment.

In addition to the **Introduction**, this manual contains the following major sections:

- **User's Guide** describes the operations available with the **MCM** software package.
- **Reference Manual** contains the description of the functions (API) and commands (ACI) .
- **Structure and Configuration of the Local Database** contains the detailed description of the local database for the motors.
- **Installation Guide** describes how to install the **MCM** software and make it ready for use.
- **Error Messages and Recovery** provides a complete list of error and diagnostic messages and possible recovery actions.

1.2 Scope

This manual describes the usage of the SW modules **mot** and **motci** as for VLT Common Software Release **`OCT98`** or higher.
<u>**Note:**</u> This release is also compatible with the VLT Common Software from Release **`DEC95`**.

The **mot** SW module is a service layer for control of VLT motor equipment on LCU.
The **motci** SW module is interfacing the **mot** SW module with WS/LCU applications via the message system.

The interface of the **mot** module to the Single Device Libraries (SDL) is **not** part of this manual and is described in [4].

1.3 Reference Documents

The following documents contain additional information and are referenced in the text.

[1] VLT-MAN-SBI-17210-0001	LCU Common Software User Manual
[2] VLT-PRO-ESO-10000-0228	VLT Programming Standards
[3] VLT-SPE-ESO-17210-0249	CCS-LCU/Motor Libraries Functional Specification
[4] VLT-MAN-ESO-17210-0776	CCS-LCU/Motor Control Module - SDL User Manual
[5] VLT-MAN-ESO-17210-0719	CCS-LCC/Common Access Interface User Manual
[6] VLT-MAN-ESO-17200-0642	VLT Common Software Installation Manual
[7] MACCON GmbH, *port* GmbH	MAC4-xxx User Manual (reshaped and translated in english)
[8] VLT-MAN-ESO-17210-0375	CCS-LCU / Driver Development Guide and User Manual
[9] VLT-MAN-ESO-17200-0981	VLT Software Problem Report Change Request User Manual

1.4 Abbreviations and Acronyms

The following abbreviations and acronyms are used in this document:

ACI / API	Application Command / Programmatic Interpreter
CCS	Central Control Software
HW / SW	Hardware / Software
LCC	LCU Common Software
LCU	Local Control Unit
DB	Local Database
MCM	Motor Control Module
SDL	Single Device Library
VLT	Very Large Telescope
WS	UNIX Work Station

1.5 Stylistic Conventions

The following styles are used:

bold in the text, for commands, file names, etc. as they have to be typed.

italic in the text, for parts that have to be substituted with the real content before typing.

`teletype` for examples.

<*name*> in the examples, for parts that have to be substituted with the real content before typing.

The **bold** and *italic* styles are also used to highlight words.

1.6 Naming conventions

This implementation follows the naming conventions as outlined in the VLT Programming Standards [2].

1.7 Problem Reporting/Change Request

The procedure to follow is described in [9] .

2 USER'S GUIDE

This section describes the API functions and ACI commands provided by the **mot** and **motci** SW modules.

2.1 Overview

The Motor Control Module is designed for a standard LCU environment, consisting of a VMEbus system with a CPU board, running the VxWorks operating system. The MCM makes extensive use of the LCU Common Software facilities, such as Event, Error and Signal Handling, Time functions and, in particular Local Database access.

The Motor Control Module forms a service layer between application level and device driver level to provide a standardized, hardware independent interface between applications and motors. An application attaches the required motor(s) and performs operations with that/these motor(s) by means of the API routines or ACI commands without knowing details about drivers or boards.
In particular, the ACI, used in combination with the **motei**, provides a mean to the engineer to tune the motor configuration parameters and test the motions without having to write any code.

The following environment must be provided and set-up at boot-time to enable operation of the Motor Control Module.
All versions correspond to VLT Common Software Release **`OCT98`**, unless specified.

- CPU board Motorola MVME167 (MC68040) or MVME2604 (PowerPC 604e)
- motor hardware and corresponding VME boards
  - DC or Stepper motor
  - Servo Amplifier
    ESO-VME4SA-X1
    ESO-VME4ST
  - Motion Controller
    MACCON-MAC4-INC, from version 4.1A (recommended is 4.2A)
    MACCON-MAC4-SSI, from version 4.1A (recommended is 4.2A)
    MACCON-MAC4-STP, from version 2.0A
  - Digital IO Board
    ACROMAG Series 948x
  - Time Interface Module
    ESO-TIM
- VLT Common Software , backwards compatible with releases from **`DEC95`**
- device drivers, one for each board type:
  - **ampl**, for the Servo Amplifier
  - **mcon**, for the Motion Controller
  - **acro**, for the digital I/O board
  - **tim**, for the Time Interface module
- MCM ACI **motci** & API **mot** modules
- MCM SDL modules (see [4])
- local database containing entries for the motors to be used, (see section 4)

Note: The version of the Motor Control Module described in this manual works **only** with the above described versions of board firmware and device drivers. It is **not** possible to run the current version of the Motor Control Module together with older versions of the drivers.

Figure 1 shows the hardware and software environment of the **MCM** :

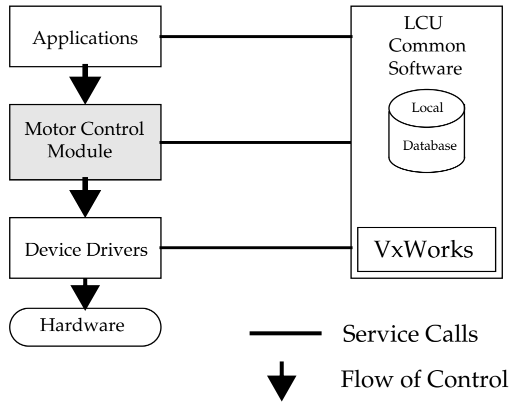


Figure 1 - Motor Control Module Environment

- The interface to the applications is covered by the modules **mot** (API), and **motci** (ACI), which are described later in this document. The API provides a library of routines callable by LCU applications, while the ACI provides a LCU process implementing a set of commands which can be sent by LCU and WS applications (see also Figure 2).
- The interface to the device drivers, through which all hardware accesses are made, is covered by the so-called SDL modules. This interface is described in [4].
- The local database is accessed by dedicated routines, which are provided by the LCU Common Software [1] and by the Common Access Interface module [5].
- The facilities of the LCU Common Software are used whenever applicable for the needs of the Motor Control Module.
- The interface to the VxWorks operating system, i.e. to VxWorks specific system services is kept as small as possible.

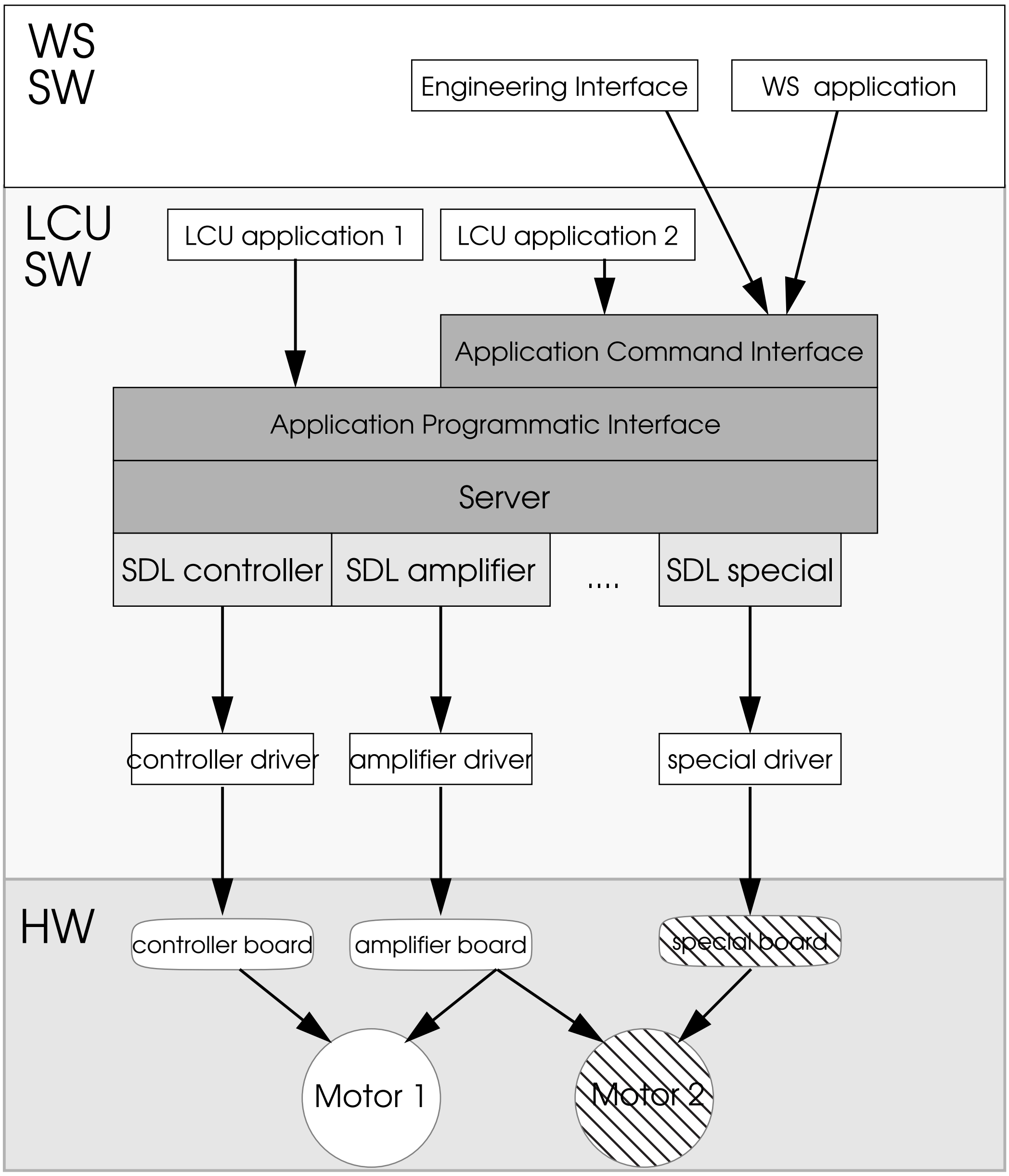


Figure 2 - Role played by the various MCM components

2.2 Supported Configurations

This section describes the hardware components and configurations from the final-version point of view. The restrictions of the implementation of this version and the explicit identification of boards are described in the next section.

The motor hardware (i.e. the logical level below the driver level) can be divided into various groups. There are:

- **LCU internal boards**:
    - motion controller
    - amplifiers
    - digital I/O boards

    - analog I/O boards
    - external encoder interfaces
- **LCU external hardware**:
    - encoders
    - limit switches
    - reference switches
- **motors**:
    - stepper motors
    - DC motors

All the supported configurations are listed in Appendix A.

- VME boards: (see Appendix A)

  The only mandatory board for motion control is the amplifier, which may not be controlled by MCM (Stand-Alone mode).

Important Note: In all the supported configurations but #21 to #23, the amplifier can be replaced by any kind of Maccon MAC4-compatible purely passive amplifier.

- Limit and reference switches: There may exist two sets of upper/lower limit switches and one reference switch. When only an amplifier controls the hardware then only one set of limit switches is allowed, otherwise each combination of one or two sets of limit switches and/or one reference switch are possible.
- Types of position encoders: serial and parallel encoders for linear and circular axis with absolute (type SSI) or incremental (type INC) coding, with and without zero pulse are supported. The use of an encoder requires a motion controller board or an external encoder interface to get the encoder value.
- Additional I/O-Signals like interlock, emergency stop, motion stop etc. can be configured. Such signals are connected to a LCU through I/O-Boards. The **mot** module uses the Common Software signal handling facilities to process such signals.

Note: Internally, the **mot** module has no practical limitation on the number of motors it can support. However, each motor occupies system resources (CPU, dynamic memory, database space, etc.) on a LCU, which limits the maximum number of motors to about 30. The MCM object modules need about 200 kBytes memory space.

In addition, each motor requires the following resources:

- approx. 20 kB database space
- about 1 kB dynamic memory, not counting the task stacks
- one device descriptor per driver
- up to three tasks, besides the application task
- up to seven (7) semaphores
- up to four (4) interrupts (depending on the HW)

In the above list the number of interrupt hooks is the strongest constraint because the hardware supports a maximum of 256 interrupts, which cannot be extended. All other resources can be extended by supplying more memory and configuring VxWorks appropriately.

2.2.1 Version Related Configurations

The MCM software supports all the configurations listed in Appendix A, but #21 and #23.
These configurations consist in:

- an amplifier (mandatory). If declared, the respective SDL is used. Otherwise, the MCM does not attempt to access any amplifier board. (see 4)
- optionally a motion controller.
- a serial incremental INC/absolute SSI encoder, connected to the controller or to an external interface board.
- optionally one set of limit switches connected to the controller.
- optionally one set of limit switches connected to the amplifier.

The MCM package supports DC and Stepper motors.

2.3 Motor Registration

2.3.1 Configuration

Depending on the way the database branch of the motor has been generated (see 5.3.1), one may need to load the configuration file (<motor>.dbcfg file) into the DB branch. The configuration files are part of the application software and generated via the Engineering tool **motei**. They shall be installed in the **`$INS_ROOT/SYSTEM/COMMON/CONFIGFILES`** path and the **`$INS_ROOT`** file system shall be NFS mounted as **`/INSROOT`** on the LCU. The function motInitDb(1) performs this initialization, as shown in the example below for the motor aliased **`M1`**:

Example:

```
ccsCOMPL_STAT status;
ccsERROR      error;

...
if ((status = motInitDb( "M1","motor1.dbcfg", &error )) == FAILURE)
     {
     /* process error */
     ...
     }
```

From the VxWorks shell, the installation of a motor is performed as shown below:

```
motInitDb("M1","motor1.dbcfg")
```

or

```
motInitDb("M1","/INSROOT/SYSTEM/COMMON/CONFIGFILES/motor1.dbcfg")
```

After this step has been successfully performed, the database branch of the motor has been initialized to the values contained in the file.
If no file name is provided, the file **`$INS_ROOT/SYSTEM/COMMON/CONFIGFILES/<alias>.dbcfg`** will be loaded (assumed it exists). It is also possible to give any valid absolute path, assumed it can be accessed via NFS from the LCU.

2.3.2 Installation

Before any operation on motors can be performed, all the motors connected to the LCU and intended to be accessed by the MCM, must be installed. The installation must be done **only once for each motor** and can be invoked either from the VxWorks Shell as a tool function, or within the application. The function **`motInstall`**() requires as parameter the alias (or the absolute path to the database branch) of the motor. It does various operations:

1. Allocate and initialize internal data structures
2. Check the database structure
3. Resolve direct addresses of database attributes for performance reasons
4. Call the SDL installation functions
5. Check and convert database information:
    a. Resolution of the external functions (see HW-Initialization and Unit conversions): At installation time, the only requirement is that all external functions are resolved (their entry points have been found in the symbol table).
    b. Unit conversion check
    c. Conversion of the database attributes in board units: Database entries specified in user units are here converted to board units for performance.
6. Configure the I/O signals

<u>Example:</u>

```
ccsCOMPL_STAT status;
ccsERROR      error;

...
if ((status = motInstall( "M1",&error )) == FAILURE)
    {
    /* process error */
    ...
    }
```

From the VxWorks shell, the installation of a motor is performed as shown below:

```
motInstall("M1")
```

The tool function **`motPrintMotors`** displays all the installed motors (see 3.3.4), see also motMotors(3).

2.3.3 Deinstallation

<u>Deregistration:</u> a motor can be de-registered from the MCM, after it has been detached, by the function **`motDeinstall`**(), which offers the same interface as **`motInstall`**().

2.4 Application Programmatic Interface

The **mot** SW module consists in two parts: the API functions and the Server routines. The API functions are the one that are invoked by the application, the Server routines manage the interface to the SDL layer; they operate on a single motor (see Figure 2).
Two groups of functions have to be distinguished among the API routines:

- motion control : these functions are designed for operating on a set of motors
- unit conversion: these functions apply for a single motor.

The motor branches of the local database contain all information needed by the API routines for motion control. The direct access to the database by applications is foreseen only for maintenance purposes (i.e. motor parameters configuration) and for the special applications (e.g. UIF) to monitor the motor status. Parameter passing from the application to the MCM is made via the arguments of the API routines. No parameter passing through the Local Database is foreseen.
Hence, during normal operation, only the API routines are invoked by applications.

The access to motors is realized by so-called *motor handles*. A motor handle is a number, which is delivered by the **mot** SW module to an application attaching a motor and enables the access to this motor. The access to a set of motors is released by detaching them.

One application can access several motors and the same motor can be accessed simultaneously (with restrictions, described below) by several applications. Moreover, an application can pass a motor handle to another application to enable its access to the corresponding motor.
On attachment request, a motor is identified by its name, which corresponds to the alias of the main point of the motor branch in the local database (see section 4).

With exception of the install, clear, attach and unit conversion functions, a list of handles terminated by the value 0 has to be passed as first parameter to each **mot** API routine.

Furthermore, the last argument of each API routine to be passed is the address of a **`ccsERROR`** structure. In the case of an error the error reason and number are passed back to the calling routine by this structure. **In general, the processing loop of the passed handle list is broken when the first error occurs**.

Finally, each API routine returns a **`ccsCOMPL_STAT`** completion status. Further arguments of API routines are function dependent. Hence, the generic synopsis of an API routine is:

```
ccsCOMPL_STAT mot<Function>(motHANDLE *handle,...,ccsERROR *error)
```

where the ellipsis **...** stands for the function dependent parameters.

The API functions available in the current version are:

1. motAttach — attach motor
2. motCheckInterlock — check interlock line
3. motCheckBrake — check brake status
4. motCheckMove — check validity of requested motion for a set of motors
5. motClampBrake — clamp brake
6. motConnect — connect motor electrically
7. motDetach — detach motor
8. motDigToCurrent — convert motor current from digital value to user specific unit
9. motDisable — disable motion controller control loop
10. motDisconnect — disconnect motor electrically
11. motEnable — enable motion controller control loop
12. motEncToIndex — convert position from encoder specific unit to a position index
13. motEncToName — convert position from encoder specific unit to a position name

14. motEncToPos	convert position from encoder specific unit to user specific unit
15. motEncToVel	convert speed from encoder specific unit to velocity reference unit
16. motGetOpMode	get actual operational mode
17. motGetParListAddress	read the address of the user data structure from the Database
18.motGetHandle	retrieve motor handle for external encoder handling
19.motGetSemIEV	retrieve semaphore for external encoder handling
20.motGetStatus	get actual motor status information
21. motInitHw	initialize motor from the HW point of view
22. motInitSampling	start motor status sampling
23. motInitSw	initialize motor from the SW point of view
24. motLoadPar	load parameter to amplifier/motion controller boards
25. motMove	move motor
26. mot NameToEnc	convert position name to encoder specific position unit
27. mot NameToVel	convert speed name to encoder specific velocity unit
28. motPosToEnc	convert position from user specific unit to encoder specific unit
29. motRefToVel	convert speed from velocity reference unit to user specific unit
30. motResetAxis	reset axis
31. motResetBoards	reset boards
32. motRetrievePar	retrieve amplifier /motion controller board parameter
33. motSetDefaultUnits	set default user units
34. motSetMotionMode	set motion mode
35. motSetOpMode	set operational mode
36.motSetParListAddress	write the address of the user data structure to the Database
37.motSetSection	set section for named positions
38.motSetSwLimits	set SW limits of a set of motors
39.motStop	stop motor
40.motStopSampling	stop status sampling task
41.motUnclampBrake	unclamp brake
42.motVelToEnc	convert speed from user specific unit to encoder specific unit
43.motVelToRef	convert speed from user specific unit to velocity reference unit
44.motWaitInit	wait for HW initialisation procedure completion
45.motWaitMove	wait for motion completion
46.motWriteEncoderValue	write encoder value to motion controller board

See section 3.1 for detailed information concerning each function.

Note 1 The priority of the calling task **is free**.

Note 2 In the following sections, various examples will be listed. The variables used will be declared with the appropriate type where they are needed. The variables, which have been previously declared, will not be declared again.
Database attributes will be referenced by <*path*>`:SUB_POINT.attribute`, where <*path*> is the path to the motor point (e.g. `:INS1:DC_MOTOR1`).

2.4.1 Access Control

To get access to a motor, an application must attach the requested motor. The function **motAttach** is the first function to call. It returns the motor handle, with which further operation is possible. The application can attach the motor in several modes:

- **motREAD_ONLY** motions are not allowed, only the request of status and configuration parameters, the status sampling, unit conversions and waiting for motion to complete are authorized
- **motEXCLUSIVE** all actions are allowed, no other application can perform motions on the motor
- **motEMERGENCY** everything allowed with highest priority (maintenance purpose, emergency case)

Only one application at a time can attach a motor in exclusive mode. Further attempts are rejected with an error. However, an arbitrary number of applications can attach in read-only mode. The emergency mode is always allowed without restrictions but shall really be used in case of emergency only.
The associated literals to the type **motACCESS_MODE** are defined in **motDefines.h**.

Table 1 shows the allowed combinations, where RO, EX and EM stand resp. for Read-Only, Exclusive and Emergency; [n] means many times, [1] once only.

from \ to	RO	EX	EM
RO	OK [n]	OK [n]	OK [n]
EX	OK [1]	FAIL	OK [1]
EM	OK [1]	OK [1]	FAIL

Table 1 - Access mode combination matrix

A connection to a motor is released, i.e. the corresponding motor handle is freed by calling the function **motDetach,** which results in the application losing access to the motor.

<u>Important Note:</u> After a motor has been attached in Exclusive mode, it must be SW and HW initialized. Detaching an initialized motor from Exclusive mode leads to loose the initialization.

<u>Example</u>:

```
....
#include "mot.h"

ccsCOMPL_STAT myExample(....,ccsERROR *error)
    {
    /* handles for motor access */
    motHANDLE     handleEX[2] = {0,0};
    motHANDLE     handleRO[2] = {0,0};
    /* database path to the motor to be attached */
    dbSYMADDRESS  motorName;
    /* motor status data structure */
```

```
motSTATUS     motorStatus;
....
ccsCOMPL_STAT status;
....

/*
 * attach motor named DC_MOTOR1 in EXCLUSIVE access mode
 * in motAttach a motor is identified by its name,
 * in any other API function, a motor is identified by
 * its handle returned by motAttach
 */
strcpy(motorName,":INS1:DC_MOTOR1");
status = motAttach(motorName,motEXCLUSIVE,&handleEX[0],error);
if (status == FAILURE)
    {
    /* process the error */
    }

/*
 * attach motor named DC_MOTOR1 in READ_ONLY access mode
 */
status = motAttach(motorName,motREAD_ONLY,&handleRO[0],error);
if (status == FAILURE)
    {
    /* process the error */
    }
...
/*
 * perform actions on the motor
 */
status = motInitSw(handleEX,TRUE,error);
if (status == FAILURE)
    {
    /* process the error */
    }
...
/*
 * get the motor status
 */
status = motGetStatus(handleRO,&motStatus,error);
if (status == FAILURE)
    {
    /* process the error */
    }
...
/*
 * get the motor status
 */
status = motGetStatus(handleEX,&motStatus,error);
if (status == FAILURE)
    {
    /* process the error */
    }
...
/*
 * detach motor
 */
status = motDetach(handleEX,error);
```

```
if (status == FAILURE)
    {
    /* process the error */
    }
status = motDetach(handleRO,error);
if (status == FAILURE)
    {
    /* process the error */
    }
...
return(status);
}
```

2.4.2 Operational Modes

Each motor can be used in several modes of operation:

The **NORMAL** mode is the standard operational mode and the only mode where motions are possible.

The **SIMULATION** mode is used for operation without hardware. No driver calls are made in this mode. The simulation mode is handled on the Single Device Library level (see [4]). The processing down to the level of driver calls is identical to the **NORMAL** mode. In simulation mode all hardware actions are simulated to be done instantaneously and successfully.

In **HANDSET** mode the motor is controlled manually through LCU boards. No motion can be started, but read access is granted for the current status.

The mode **NOT AVAILABLE** signals that a motor cannot be used. This state is entered by command after a malfunction is detected from the outside. No motion is allowed. Read access is granted for the current status.

The mode **FIXED POSITION** is internally treated in the same way as the **NOT AVAILABLE** mode. It is just for information of an application that the motor is available but cannot be moved.No motion is allowed. Read access is granted for the current status .

The operational mode can be set by an application through the **`motSetOpMode`** routine. It can be used regardless of the active operational mode. The associated literals to the type **`motOPMODE`** are defined in **`motDefines.h`**.It is not allowed to change the operational mode while a motor is moving.
The current operational mode can be requested through a **`motGetOpMode`** call.

Furthermore, the current operational mode is stored in the local database attribute <*path*>`:STATUS.opMode`.

On the first attachment, an initial operational mode is read from the database and activated:
attribute <*path*>`:SERVER:CONF.motor(0,opMode)`.

Note : When a motor is not anymore attached by any application, its operational mode is set to the internal mode **`motDETACHED`**. This mode can not be selected by the application.

Note : When the operation mode of a motor is changed by an application, its initialization state is reset to **`motNOT_INIT`**.

2.4.3 Initialization Procedure

After the first attach in exclusive mode, a motor must be initialized before any motion can be performed.
The initialization is divided into two phases: software and hardware initialization.
The software initialization must be performed first. The motor configuration as stored in the database is loaded to the boards and the internal data structures are initialized.

Example:

```
vltLOGICAL mandatory;

mandatory = TRUE;
if ((status = motInitSw(handleEX,mandatory,error)) == FAILURE)
    {
    /* process the error */
    }
```

If the logical variable ***mandatory*** is **TRUE**, the SW initialization will be performed in any case; if it is **FALSE**, the SW initialization will be performed only if the motor has not been initialized before.
On success, the attribute *<path>*`:STATUS.initStatus` is set to **motSW_INIT**. The associated literals to the type **motINIT_STATUS** are defined in **motDefines.h**. On failure, it is set back to **motNOT_INIT**.

The SW initialization completes when all parameters used by the user-defined functions are valid (see 2.4.12).

In the case the encoder is not connected to the Motion Controller board and the position information is available from the CPU (valid VME address), a special procedure must be implemented that updates the position information as requested by the Motion Controller (assumed the board can handle this feature; available on **MACCON MAC4** from this release on). Detailed information is given in section 2.4.13.
This procedure must be invoked before any motion is initiated.

The hardware initialization performs a sequence of actions (motions and settings), necessary to bring the motor and its parameters to a known initial state. The wanted sequence must be defined in the LCU Local Database, attribute *<path>*`:SERVER:INIT.actions`. The number of actions is not limited. The associated literals to the type **motACTION_TYPE** are defined in **motDefines.h**:

- move the motor
- clamp the brake
- disconnect the motor
- initialize the actual position value
- set lower/upper software limits
- invoke a user-defined initialization procedure
- invoke a on-board HW initialization procedure
- delay the nex step

An action is defined by the following parameters:

1. actionType type of action (last entry must be **motACTION_END**)
2. userFctName name of the user-defined HW initialization procedure (see section 2.4.12)
3. userParList address of a data structure, which may be required by the user-defined function

4. initCode on-board HW initialization procedure number/delay in milliseconds
5. motionMode motion mode (see section 2.4.4)
6. position position to move to (see section 2.4.5)
7. posType positioning type (see section 2.4.5)
8. speed definition of the speed (see section 2.4.5)

A typical sequence of actions to initialize a motor could be

- move to reference switch
- initialize actual position value
- define upper and lower software limit
- move to a predefined starting-point

Example:

```
mandatory = TRUE;
if ((status = motInitHw(handleEX,mandatory,error)) == FAILURE)
    {
    /* process the error */
              ...
    }
```

If the logical variable mandatory is **TRUE**, the HW initialization will be performed in any case; if it is **FALSE**, the HW initialization will be performed only if the motor has not been initialized before (i.e. the attribute <*path*>:STATUS.initStatus is **motSW_INIT**). While the HW initialization is running, the attribute initStatus is set to **motHW_INIT_RUN**. On success, the attribute initStatus is set to **motINIT_DONE**; on failure, it is set back to **motSW_INIT**.

The HW initialization completes when positions which have been defined relative to HW positions are fully resolved.

The HW initialization procedure may last some time, and the application may want to use this time for other operations. It would also like to know exactly when the HW initialization procedure is completed, without consuming CPU time. The function **motWaitInit** fulfills these expectations:

Example:

```
vltINT32 timeout = 300000;  /* in milli-seconds */
status = motWaitInit(handle,timeout,&motStatus,error);
if (status == FAILURE)
    {
    if (error->errorNumber == motERR_TIMEOUT)
        {
            /* process the timeout */
            ...
        }
        else
        {
            /* process the error */
            ...
        }
    }
```

The function returns when all the motors of the handle list have completed their HW initialization procedure or a timeout has occurred, or immediately if an error occurred. The function returns **SUCCESS** when the HW initialization has been successfully completed within the given time; it returns **FAILURE** if a timeout has occurred (error number set to **motERR_TIMEOUT**), or if the procedure failed (the error number is then set appropriately).
The status of each specified motor is updated on function completion, regardless of the returned completion status.
On failure, the HW initialization procedure of the remaining motors is not interrupted.
Only one process can wait for the HW initialization procedure of a given motor to complete.
The timeout can also be set to **WAIT_FOREVER** (-1), this should be handled with care.

2.4.4 Motion Modes

In order to be able to move a motor, a motion mode must be set through the **motSetMotionMode** routine. Possible choices:

In **motPOSITION_MODE** mode, a motor can be moved to a position defined by a relative or absolute value.

In **motDIRECT_MODE** mode, a motor will be moved directly **via the amplifier**, whereby only the speed can be specified. The speed can be changed at any time by a further call to motMove.

In **motTRACKING_MODE** mode, a motor can be moved to a given absolute or a relative position, or at a given speed, whereby the next position/speed can be redefined at any time during the motion by a further call to motMove.

In **motSPEED_MODE** mode, a motion is defined by a speed and **controlled by a motion controller**. After starting a motion by motMove, the motor will move with the specified speed until a limit is reached or the motor is stopped by motStop. The speed can be changed at any time during the movement by a further call to motMove.

The associated literals to the type **motMOTIONMODE** are defined in **motDefines.h**.

Example:

```
motMOTIONMODE motionMode;
motMOTION     motion;

motion.posType = motABSOLUTE;
motion.pos.how = motBY_INDEX;
motion.pos.index = 0;
motion.speed.how = motBY_NAME;
strcpy(motion.speed.name,"coarse");
motion.lastStep = TRUE;

/* perform an absolute motion to the named position indexed 0 */
/* in POSITION mode at the named speed "coarse"                */
motionMode = motPOSITION_MODE;
status = motSetMotionMode(handle,&motionMode,error);
if (status == FAILURE)
    {
    /* process the error */
    }
status = motMove(handle,&motion,error);
if (status == FAILURE)
    {
    /* process the error */
    }
```

2.4.5 Motion Specification

A motion is specified via a **motMOTION** structure. A sequence of motions is specified by an array of such structures. The **motMOTION** structure type is defined in the motor control module include file **mot.h**. A motion structure consists of the following parameters:

- the position type
- the position specification
- the speed specification
- a flag indicating, when **TRUE**, that this structure is the last element of the motion sequence array.

The **position type** specifies how to interpret the specified position. The associated literals to the type **motPOS_TYPE** are defined in **motDefines.h**. Possible types are:

- **motABSOLUTE** The position value defines an absolute position.
- **motRELATIVE** The position value defines a relative position.
- **motUPPER_LIMIT** The position is the upper HW limit.
- **motLOWER_LIMIT** The position is the lower HW limit.
- **motREFERENCE_SWITCH** The position is the Reference Pulse
- **motINDEX_PULSE** The position is the Index Pulse.
- **motHOME_POSITION** The position is the Home position.
- **motBY_SPEED** The motion is defined merely by a speed.
- **motTWO_STEP_ABS** The motion is a two step absolute motion
- **motTWO_STEP_REL** The motion is a two step relative motion
- **motHALF_TURN_ABS** The motion is a two step absolute motion of half a turn
- **motHALF_TURN_REL** The motion is a two step relative motion of half a turn

Up to 10 steps may be defined in a sequence, but the position types **motINDEX_PULSE**, **motTWO_STEP_ABS**, **motTWO_STEP_REL**, **motHALF_TURN_ABS** and **motHALF_TURN_REL** extend to 2 entries, increasing the length of the sequence. If the sequence is longer than 10 entries, an error message will be logged (**motERR_EXTRA_IGNORED**), and the first ten entries of the sequence will be processed.

One additional position type is available for the HW Initialization only: **motOFF_REFERENCE_SWITCH** which operates in **POSITION** mode. The position of the motor is checked against the Reference Switch; if the Reference switch is not active, no motion is initiated, otherwise the motor is moved of the specified relative displacement off the Reference Switch.

For relative positioning, only positions specified by value are allowed (see below).

According to the definition of the motion modes, table 2 shows the possible combinations of motion mode and position type:

Position Type / Motion Mode	Motion Absolute / Relative	Motion to Up / Low HW Limits	Motion to Index Pulse, Home & Reference Switch	Motion By Speed	Two Step Half Turn Absolute / Relative
motSPEED_MODE	No	Yes	Yes	Yes	No
motDIRECT_MODE	No	Yes	No	Yes	No
motPOSITION_MODE	Yes	Yes[a]	Yes[a.]	No	Yes
motTRACKING_MODE	Yes	No	No	Yes	No

Table 2 - Motion mode vs. Motion type matrix

a. Access via named positions table only (assumed these positions have been defined by name) or as relative motion from these positions (assumed the absolute position of these positions is known by the system).

The motion to the index pulse is performed in two steps: the first step is a coarse search made at the given speed, the second is made in reverse direction at the speed predefined in the database attribute <*path*>`:SERVER:CONF.indexPulseSpeed`. Depending on the motion controller board, this feature may not be supported; however, the **mot** SW module invoke twice the same call to the motion controller **MoveToInd** (see [4]), with successively the two speeds. The way the motion controller handles these calls is not in the scope of this manual.

The motion to the Home position is defined as being the overlap of the Reference Switch and of the Index Pulse of the encoder (this position can be reached only if the hardware has been configured so that these two positions overlap and assuming that the encoder delivers the Index Pulse signal (Incremental encoder only).

Two step motions are handled as normal steps in **POSITION** mode, but the motion is executed in two steps, whereby the second step is predefined in the database attribute <*path*>`:SERVER:CONF.twoStepOffset`. This offset is added to the target position for the first step, then the way is covered back at the predefined speed. This feature allows the positioning to be executed always from the same side of the target position (eliminating possible backlash effects).

Half Turn motions are handled as normal steps in **POSITION** mode for **CIRCULAR** axis only, but the motion is executed in two steps, whereby the second step is predefined in the database attribute <*path*>`:SERVER:CONF.twoStepOffset`. This absolute offset is subtracted to the target position for the first step, then the way is covered in the same direction at the predefined speed. This feature allows the positioning to be executed in a deterministic way.
However, for not optimized circular axis, the motor will not (per definition) cross the zero.

If the requested position (HW limit, Reference Switch, Index Pulse or Home position) is not available on the hardware or the action is not supported by the firmware/driver or SDL, the motion fails (see section 2.4.6).

The **position** and the **speed** can be specified in three, resp. four ways:

- **by default**, whereby the default speed is taken (index in named speed table).

- **by value**, whereby all information defining the position or the speed has to be provided by the user.
- **by name**, whereby only a position/speed name has to be provided by the user and any further information is retrieved from the database attributes:
  <*path*>`:SERVER:CONF.namedPositions/Speeds`, containing up to 15 entries.
- **by index**, whereby only an index to the namedPositions/Speeds attribute of the motor in the local database has to be provided by the user and any further information is retrieved from the local database

The associated literals to the type **`motSPEC_TYPE`** are defined in **`motDefines.h`**,.

The following parameters have to be defined by the user specifying the target position or speed:

- **by value**: The position or speed is specified by a value and a unit (see data structure type **`motMETRIC`** in **`mot.h`**). For the speed, three additional parameters are required (see section 2.4.7 for explanation):
  - pollInt Monitoring polling interval in ms of the *far* phase; the motor is moving to the target position
  - endPollInt Monitoring polling interval in ms of the *near* phase; the motor is in the target window
  - inPosTime Needed in-position time in ms for motion complete condition.

Example:

```
....
motMOTION motion;
.....
/*
 * specify position by value
 * next position to move 10 deg
 */
motion.posType = motABSOLUTE;
motion.pos.how = motBY_VALUE;
motion.pos.number.value = 10.0;
strcpy(motion.pos.number.unit, "deg");
...
/*
 * specify speed by value
 * requested speed 1deg/sec
 * monitoring interval "far" phase 1000ms
 * monitoring interval "near" phase 100ms
 * in-position time interval 1000ms
 */
motion.speed.how = motBY_VALUE;
motion.speed.number.value = 1.0;
strcpy(motion.speed.number.unit, "deg/sec");
motion.speed.pollInt = 1000;
motion.speed.endPollInt = 100;
motion.speed.inPosTime = 1000;

/*
 * specify position by value
 * 10 deg below the upper HW limit
 */
```

```
motion.posType = motUPPER_LIMIT;
motion.pos.how = motBY_VALUE;
motion.pos.number.value = -10.0;
strcpy(motion.pos.number.unit, "deg");
```

- **by name**: for the position, an additional parameter is required specifying the offset to the named position: the so-called section (the offsets are defined for each namedPosition entry in the database). The associated literals to the type **motSECTION** are defined in **motDefines.h**:
  - **motOBSERVATION** position as defined
  - **motMAINTENANCE** observation position + maintenance offset.
  - **motUSER_DEFINED** observation position + user offset

  Example:

```
....
motMOTION motion;
.....
/*
 * specify position by name
 * next position to move is "RED", Maintenance position
 */
motion.posType = motABSOLUTE;
motion.pos.how = motBY_NAME
strcpy(motion.pos.name.posName, "RED");
motion.pos.name.section = motMAINTENANCE;
....
    /*
     * specify speed by name
     * moving speed is "FAST"
     */
    motion.speed.how = motBY_NAME
    strcpy(motion.speed.name, "FAST");
```

- **by index**: The index number specifies the record number of the entry of the attribute named Positions/Speeds to be used.

  For the position indexing, it is possible to specify the section (as for by name). The following macros are available:

  `motSET_OBSERVATION_INDEX(index)` to access the Observation section

  `motSET_MAINTENANCE_INDEX(index)` to access the Maintenance section

  `motSET_USERDEFINED_INDEX(index)` to access the User-Defined section

  The default section is Observation (for backwards compatibility).

  Note: The first entry of a table corresponds to the record number 0.

  Example:

```
....
motMOTION motion;
.....
/*
 * specify position by index
 * the requested position corresponds to the named position
```

```
 * specified by the 4. entry (= record number 3) in the named
 * position table, additionally in the MAINTENANCE section
 */
motion.posType = motABSOLUTE;
motion.pos.how = motBY_INDEX;
motion.pos.index = motSET_MAINTENANCE_INDEX(3);
....
/*
 * specify speed by index
 * the requested position corresponds to the named speed
 * specified by the 2. entry (= record number 1) in the named
 * speed table
 */
motion.speed.how = motBY_INDEX;
motion.speed.index = 1;
....
```

Important Note:

For the position specification by name or by index, the value stored in the field `number` is taken as an offset if the string defining the unit is not empty.

Example:

```
....
motMOTION motion;
.....
/*
 * specify position by name + offset
 * next position to move 10 deg above the named position
 *  "RED" in Maintenance section
 *
 */
motion.posType = motABSOLUTE;
motion.pos.how = motBY_NAME
strcpy(motion.pos.name.posName, "RED");
motion.pos.name.section = motMAINTENANCE;
strcpy(motion.pos.number.unit,"deg");
motion.pos.number.value = 10.0;
....
```

2.4.6 Motion Control

Before trying to move a motor the following actions have to be performed:

- Attach the motor in **EXCLUSIVE** mode, calling **motAttach** (see section 2.4.1).
- Perform the SW and HW initialization of the motor, calling **motInitSw** and **motInitHw** (see section 2.4.3)
- Configure the motion mode according the intended movement, calling **motSetMotionMode** (see section 2.4.4).

After successful execution of these operations a single motion or sequence of motions of the motor can be started using the **motMove** function.

Motions are rejected, when the access mode is not **EXCLUSIVE** or **EMERGENCY**, the motor is not available because of hardware malfunctions, the brake is clamped, the interlock line is active or the initialization procedure failed.

An application can ask once or periodically for the status of a motor or a set of motors, by calling the routine **motGetStatus** (see also 2.4.8), or wait for the motion completion, by calling the routine **motWaitMove**, which works like **motWaitInit**. This function returns either on motion successful completion, time-out or abnormal event.

Two types of timeout may occur:

Motion step timeout	The motion step could not be completed within the time within which any motion step shall complete. This time is defined in the database attribute <*path*>`:SERVER:CONF.motor(0,timeout)` in milliseconds.
Motion sequence timeout	The motion sequence could not be completed within the time given to the motWaitMove function.

This routine returns a structure, containing the status of the motors. The associated literals to the type **motMOTION_STATUS** are defined in **motDefines.h**:

- **motSTANDING** The motor has reached the soll-target position,
- **motMOVING** The motor is moving,
- **motTIMEOUT** The motion is aborted due to timeout,
- **motABORTED** The motion is aborted due to an error, or has been interrupted,
- **motREDEFINED** The motion has been redefined while moving.

Example:

```
#include "mot.h"

ccsCOMPL_STAT myMoveExample (....,ccsERROR *error)
{
....
motHANDLE      handle[2] = {0,0};
motMOTION      motion;
motMOTIONMODE  motionMode;
motSTATUS      motStatus;
....
/* attach motor/fill motionSequence data structure */
status = motAttach("DC1",motEXCLUSIVE,handle,error);
....
/* initialize motor SW/HW */
status = motInitSw(handle,TRUE,error);
....
status = motInitHw(handle,TRUE,error);
....
/* fill motion sequence data structure */
motion.posType = motABSOLUTE;
motion.pos.how = motBY_INDEX;
motion.pos.index = 0;
motion.speed.how = motBY_NAME;
strcpy(motion.speed.name,"coarse");
motion.lastStep = TRUE;

/* perform an absolute motion to the named position indexed 0
 * in POSITION mode at the named speed "coarse" */
motionMode = motPOSITION_MODE;
```

```
if (motSetMotionMode(handle,&motionMode,error) == FAILURE)
    {
    /* process the error */
    ...
    }

/* initiate movement */
if (motMove(handle,&motion,error) == FAILURE)
    {
    ....
    }
 /*
  * wait for end of movement 10000 ms
  */
 if (motWaitMove(handle,10000,&motStatus,error) == FAILURE)
     {
     if (error->errorNumber == motERR_TIMEOUT)
         {
         /* time-out reached */
          ....
         }
     else
         {
         /* an error has occurred */
         ....
         }
     }
/* movement successfully completed */
 ....
return(status);
}
```

An application can stop a motor by using the **motStop** routine. After the motor has stopped, another process waiting with **motWaitMove** is resumed and returns **FAILURE**. The motion status is set to **STANDING**; if in **POSITION** motion mode, the status flag **inPosition** is not set because the target position was not reached.

2.4.7 Motion Monitoring

From the point of view of the MCM a movement consists of two phases:

- the *far* phase, where the motor drives to the target position
- the *near* phase, where the target is almost reached, the motor is in the target window and the motor slowly moves to its end position

A movement is automatically supervised by a monitoring task, which periodically requests and evaluates the actual status. Three parameters have to be made available to the monitoring task:

- the polling interval in ms for the *far* phase
- the polling interval in ms for the *near* phase
- the value specifying the required "**in position**" time in ms for the motion complete condition

These parameters have to be specified either by the user via the **motMOTION** data structure passed to **motMove** (see section 2.4.5) or, if a named speed is used, in the corresponding namedSpeeds record of the local database.

For each motor a time-out value is configured in the local database, to specify when motions of this motor shall be treated as lasting too long and be timed-out. However, the motion monitor task does not undertake any actions on abnormal events or time-out but informs the application by setting the motor status in the local database accordingly.

If the motion completion is to be reported by an interrupt (see 2.4.10), it returns the motion status **`STANDING`** as soon the motion end event has been registered by the motion monitor (assumed the in Position time is larger than the target settle time, see motion controller **mac4** parameter **`TT`**).

2.4.8 Status Sampling

The Motor Status Sampling facility performs monitoring and status sampling of any set of motors, no matter whether they are moving or initialized. The sampling is started by the routine **`motInitSampling`**. A sample period can be specified at which the general status data in the local database is updated. The **`motStopSampling`** routine is used to stop the sampling. For detailed information about the general status data section of a motor in the local database, see section 4.

The status sampling task is automatically spawned after the motor has been successfully SW initialized. The sampling task updates the status with the period specified in the database attribute <*path*>`:SERVER:CONF.motor(0,timemon)` in milli-seconds. The sampling task runs until the motor is detached or explicitly stopped by **`motStopSampling`**. The sampling task remains alive (regardless of any call to **`motStopSampling`**) as long a motor is at least SW Initialized. For Read-Only processes, calls to **`motStopSampling`** will have little effect.

Note: During all the time a motor is SW initialized, the status part of the database is automatically updated on change of any value, but the quantities defined in user units (speed, current, position, next position, position names and following error).
These fields are updated each time **`motGetStatus`** in invoked, thus when the status sampling is running.

The section in which the named position will be retrieved from is set by the section passed in the motion definition. If no section is specified, the actual section remains valid (target defined by value). The function **`motSetSection`** allows the setting of a new section, it remains valid until a motion referencing another section is initiated. It is however possible to invoke **`motGetStatus`** specifying a different section: the DB values will continue to be updated according to the actual section, but the returned status structure will be updated according to the section set in `status.posSection` (the section used for the `nextPos` fields is the same).

2.4.9 Unit Conversion Facility

The Motor Control Module allows the user to specify positions, speeds and electric currents in common user units (e.g. meters, meters/second or ampere). Internally, such values must be known in board specific units (e.g. encoder ticks, digital binary or velocity reference). Vice versa, the boards return run-time data in specific units which must be converted into user units.

Therefore, the **mot** SW module provides a set of unit conversion functions, which are used internally but also accessible from application level. The unit conversion facility is designed to transform user units for position, speed and electric current into board specific values and vice versa.

The **mot** API provides the following routines for unit conversion:

- **motPosToEnc** - convert position user unit to encoder values
- **motEncToPos** - convert encoder values to position user unit
- **motNameToEnc** - convert position name to encoder values
- **motEncToIndex** - convert encoder values to position index
- **motEncToName** - convert encoder values to position name
- **motNameToVel** - convert speed name to encoder values
- **motVelToEnc** - convert speed user unit to encoder values
- **motEncToVel** - convert encoder values to speed user unit
- **motVelToRef** - convert speed user unit to velocity reference
- **motRefToVel** - convert velocity reference to speed user unit
- **motDigToCurrent**- convert digital value to current unit

An application can specify a position, speed or current argument to a **mot** function in any appropriate user units. A user unit is of the type **motUNIT** and can be seven characters long at maximum. The speed unit **must** contain a slash '**/**'. Possible units are for example:

- "deg" for a position
- "deg/sec" for a speed
- "mA" for the motor current

The function **motSetDefaultUnits** allows the user to set the three user units to be used for every conversion from board to user unit. For each user unit, a conversion method must be defined in the local database motor section, to be able to convert a speed/position or motor current value to a board specific value and vice versa (see section 4). The board specific values and their units are:

- for a position value: encoder ticks ("**Enc**")
- for a speed value: encoder ticks per ms ("**Enc/ms**") for a motion controller board and velocity reference value ("**Ref**") for an amplifier board
- for a motor current value: a digital motor current value ("**Dig**")

The board units are defined in **motDefines.h**: **motBOARD_UNIT_POS**, **motBOARD_UNIT_SPD**, **motBOARD_UNIT_REF** and **motBOARD_UNIT_CUR**, resp. for position, speed (Motion Controller and Servo Amplifier related) and current.

For each motor all necessary user unit conversion methods have to be defined in the local database motor section, point <*path*>:SERVER:UNIT. This point contains the unit conversion table named "**unitConversion**", whereby each record defines a unit conversion method. The table must contain at least 4 methods, corresponding to the four default units. The name of the unit conversion is specified in the first field and is built up in the following way

unit conversion name = <from unit>To<to unit>

So, for example, the first field of the record specifying the unit conversion "deg" to "encoder ticks" (**Enc**) contains "**degToEnc**".

Thus for each user unit to be used for motions, the following unit conversions must be specified in the unit conversion table.

- for a position user unit
  - <user unit>ToEnc (e.g. "degToEnc")
- for a speed user unit
  - <user unit>ToEnc/ms (e.g. "deg/secToEnc/ms")

- <user unit>ToRef (e.g. "m/sToRef")
- for a user unit of the motor current
  - <user unit>ToDig (e.g. "mAToDig")

Three methods for conversion are available:

- **`motLINEAR`** linear conversion

  two parameters (slope and offset) define the conversion formula
- **`motINTERPOLATION`** linear interpolation

  a vector of 50 pairs of coordinates (x,y) defines the domain of conversion.

  the point coordinate x (or y) greater than or equal to the origin value x (or y) is searched, the interpolation is performed linearily between this point and its predecessor.
- **`motEXTERN`** by a user function (see 2.4.12)

Any position, speed or motor current is specified by a **`motMETRIC`** data structure containing

- the value <structure>.value (**`vltDOUBLE`**)
- the unit <structure>.unit (**`motUNIT`**)

Beside the unit conversion, the functions **`motNameToEnc`**, **`motEncToIndex`** and **`motEncToName`** allow to convert a named position to an encoder value and vice versa. Each named position to be used has to be defined in the "named positions table" in the configuration section of the local database, attribute *<path>*`:SERVER:CONF.namedPositions`.

The function **`motNameToVel`** provides the facility to convert a named speed into its value in `Enc/ms`. Each named speed to be used has to be defined in the "named speeds table" in the configuration section of the local database, attribute *<path>*`:SERVER:CONF.namedSpeeds`.

For relative positions, it is mandatory to give the position unit either as `Enc` or as the default position unit.

2.4.10 Emergency Handling and Abnormal Events

Abnormal events are caused and detected by hardware components. Up to 32 events may be configured. They can be processed by software, interrupt, signal or not processed. (Literals are defined type in **`motDefines.h`**). The database attribute *<path>*`:SERVER:CONF.events` contains the description of the event handling mechanism for all the supported events.

Interrupts are generated by the hardware. As configured in the local database, the internal interrupt service routine (**`motsrvIsr`**) can be connected to the given board in order to catch the interrupts, and inform the motion monitor about the event. If the event is to be connected to an interrupt, then the board generating this interrupt (Literals for the type **`motBOARD_TYPE`** in **`motDefines.h`**) and the name of the interrupt must be supplied.

The boards return status data which may contain information about abnormal events. This includes on-limit flags, board failure status, over-temperature and over-current. These data are stored in the local database STATUS section of each motor and updated on every **`motGetStatus`** call. The motion monitor calls this routine in every polling cycle and processes abnormal event information directly.

When an abnormal event is detected, the motion monitor writes error information into the local database `STATUS`, releases a **`motWaitMove`** or **`motWaitInit`** immediately and terminates. The application gets informed by **`motWaitMove/Init`** returning and the appropriate motion/init status in the motor status.

In case of emergency, a supervising task can get access to a motor by attaching it in EMERGENCY mode. This mode overrides even the EXCLUSIVE mode and enables all actions for the attaching task.

2.4.11 I/O Signals

Three digital I/O signals may be handled for a motor:

- Interlock Line — The status of this line may be retrieved by the function **`motCheckInterlock`**, it is also updated in the status by the function **`motGetStatus`**. If this line is active, the motor is not accessible by other applications, even if it is detached or attached in read-only mode.
  (database attribute <*path*>`:STATUS.interlock`)
- Brake Status — The status of this line is updated in the status by the function **`motGetStatus`**. If this line is active, the motor cannot be moved.
  (database attribute <*path*>`:STATUS.brakeClamped`)
- Brake Line — The brake may be clamped or unclamped by the functions **`motClampBrake`** and **`motUnclampBrake`**. The brake status of the database is updated accordingly.

These signals are handled by the digital I/O board ACROMAG. They are mapped by one I/O line per signal (fully configurable: digital I/O device, line number and logical level) as described in the attribute <*path*>`:SERVER:CONF.signals`.

2.4.12 User Function Hooks

Two types of user-defined function hooks are available:

- in the HW initialization procedure,
- for the unit conversion facility.

A user-defined function may be invoked automatically from the API while performing the HW initialization procedure or for unit conversion, if the following requirements are satisfied.

2.4.12.1 HW Initialization Procedure

An action of type **`motACTION_USER_DEFINED`** must be created in definition of the HW initialization procedure in the database (see section 2.4.3) and the following two parameters have to be set:

- userFctName — is the name of the function to be invoked,
- userParList — is the address of a data structure to be passed to the function. Depending on the function, this parameter may be optional and the structure and content of the parameter list is under responsibility of the user.

The interface of the user-defined initialization functions must comply the following rule:

```
ccsCOMPL_STAT userInitFct(void *parList, ccsERROR *error)
```

2.4.12.2 Unit Conversion

A conversion method of type **motEXTERN** must be created in the database (see section 2.4.9) and the following two parameters have to be set:

- userFctName is the name of the function to be invoked,
- userParList is the address of a data structure to be passed to the function. Depending on the function, this parameter may be optional and the structure and content of the parameter list is under responsibility of the user.

The interface of the user-defined conversion function must comply the following rule:

```
ccsCOMPL_STAT userConvFct(motMETRIC *from, motMETRIC *to,
                          void *parList, ccsERROR *error)
```

Note : The function **must** provide the conversion in both directions.

2.4.12.3 Arguments Passing Mechanism

It is possible to pass the address of a user-defined data structure to the function. The addresses of the structures must be stored into the database **before** the function is invoked.
The enumeration type **motUSER_FCT** provides the necessary literals specifying the attribute to be accessed:

motUSER_INIT for <*path*>`:SERVER:INIT.actions`, and

motUSER_CONV for <*path*>`:SERVER:UNIT.unitConversion`.

For this purpose, two functions are available **motSetParListAddress**, that stores the address of the user-defined data structure into the database, and **motGetParListAddress**, that retrieves the address from the database.

<u>Example for the HW Initialization Procedure:</u>

Assumed one entry of the HW Initialization procedure is set as follows in the INIT part of the database:

- Action **motACTION_USER_DEFINED**
- FctName "**myHwInitFunction**"

The function expects the address of a structure containing the two parameters `handle` and `param`, as declared in the structure defined below:

```
typedef struct
    {
    motHANDLE  handle;
    vltUINT16  param;
    } mtPARLIST;
```

The User HW Init function is:

```
ccsCOMPL_STAT  myHwInitFunction( void *parList, ccsERROR *error )
    {
    TparList *list  = parList;
```

```
motsrvHANDLEDATA *pHandle;     /* pointer to handle data      */
motDESCRIPTION   *desc;        /* pointer to motor descriptor */

/*
 * get pointer to motor descriptor
 */
pHandle = (motsrvHANDLEDATA *)(list->handle);
desc = pHandle->motDesc;

mtPrintf("\nUser Init H/W Function : motor '%s', param = %i\n",
         desc->motName,list->param);
return(SUCCESS);
}
```

The main code invoking the HW initialization will then be as follows:

```
TparList  myParList;
...
myParList.handle = myHandle[0];
myParList.param = (vltUINT16)tickGet();
if (motSetParListAddress(myHandle[0],motUSER_INIT,1,"myHwInitFunction",
                         (vltUINT32)&myParList,error) == FAILURE)
        return(FAILURE);
if (motInitSw(myHandle,TRUE,error) == FAILURE)
    return(FAILURE);
if (motInitHw(myHandle,TRUE,error) == FAILURE)
    return(FAILURE);
if (motWaitInit(myHandle,myTimeout,&motStatus,error) == FAILURE)
    return(FAILURE);
...
```

Important note: The user parameter list must be set to a valid address or to **`NULL`** before invoking the user function. For the user initialization function before invoking **`motInitHw`**, for the user defined unit conversion function before invoking the function. It also strongly recommended to preset these addresses after installing a motor, at the latest before initializing the motor. The address is checked for validity when set (see **`motSetParListAddress`**), **`NULL`** is always accepted.

2.4.13 External encoder support

For the case, where the encoder is not connected directly to the standard motion controller and where the CPU shall provide the encoder value, a couple of dedicated actions must be initiated so that the CPU and the Motion Controller board get synchronized.
The attributes **`SERVER:CONF.encoder(0,board)`** must be set to **`motCPU_BOARD`** and **`SDL:MAC4:CFG.configValues(IEV) = 192`** (**`0xC0`**).
During the SW initialization, the motion controller is configured to deliver an interrupt every 2.5ms. On the CPU, the Interrupt Service Routine **`motsrvIEV`** is connected to this interrupt. It catches the interrupts and gives a semaphore. A dedicated task is to be spawned that waits on this semaphore, reads the encoder information (not in the scope of MCM) and writes the new encoder value to the Motion Controller.
The task must be spawned before the Software Initialization has been successfully performed.

Example:

```
#include "mot.h"
```

```
/* Address where to read the encoder value */
vltINT32 *<mod>EncAddr = 0xF0AAAE04;

ccsCOMPL_STAT <mod>Main ( char *motor, ccsERROR *error )
    {
    motHANDLE handle[] = {0,0};

    /* Get handle of the motor, attached in EXCLUSIVE mode */
    if (motGetHandle(motor,&handle[0],error) == FAILURE)
        {
        /* process error */
        return FAILURE;
        }
    /* Spawn Encoder Task */
    if (taskSpawn("<mod>EncoderTask",70,0x18,20000,<mod>EncoderTask,
                  handle[0],0,0,0,0,0,0,0,0,0) == ERROR)
        {
        /* process error */
        return FAILURE;
        }
    /* Perform SW Initialization */
    if (motInitSw(handle,TRUE,error) == FAILURE)
        {
        /* process error */
        return FAILURE;
        }
    /* Perform HW Initialization */
    if (motInitHw(handle,TRUE,error) == FAILURE)
        {
        /* process error */
        return FAILURE;
        }
    ...
    etc...
    ...
    return(SUCCESS);
    }
```

Two functions are provided that:

- retrieves the handle of the motor, attached by another task: **motGetHandle**
- retrieves the address of the binary semaphore: **motGetSemIEV**
- writes the encoder value to the Motion Controller board: **motWriteEncoderValue**

The motor must be attached in **motEXCLUSIVE** mode.

Example:

```
#include "mot.h"

/* Address where to read the encoder value */
extern vltINT32 *<mod>EncAddr;

ccsCOMPL_STAT <mod>EncoderTask ( char *motorName )
    {
    motHANDLE handle;
    SEM_ID    encSem;
    vltINT32  encVal;
    ccsERROR  error;
```

```
/* Register to LCC */
if (ccsInit(taskName(0),0,NULL,NULL,&error) == FAILURE)
    {
    /* process error */
    return FAILURE;
    }
/* Get handle */
if (motGetHandle(motorName,&handle,&error) == FAILURE)
    {
    /* process error */
    return FAILURE;
    }
/* Get semaphore */
if (motGetSemIEV(handle,&encSem,&error) == FAILURE)
    {
    /* process error */
    return FAILURE;
    }
/* loop forever */
for (;;)
    {
    if (semTake(encSem,WAIT_FOREVER) == ERROR)
        {
        /* process error */
        return FAILURE;
        }
    /* read encoder value */
    encVal = *(vltINT32 *)<mod>EncAddr;
    /* Give the value to the motion controller */
    if (motWriteEncValue(handle,encVal,&error) == FAILURE)
        {
        /* process error */
        return FAILURE;
        }
    }
}
```

When the motor is detached or set in another operational mode than **motNORMAL**, the interrupt is disabled. The semaphore is deleted, when the motor is de-installed. The task should terminate before the motor is detached (device is closed, handle is not valid any more).

2.5 Application Command Interface

The Application Command Interface (ACI) is implemented in the module **motci** and is available as a LCU process (**motServer**). It is based on the LCC Command Interpreter, implementing in form of commands most of the functions of the Application Programmatic Interface (API). The ACI is built on top of the API and adds two major advantages:

- It allows applications running on WS or LCU to deal directly with motors independently on the platform.
- It provides, in combination with the CCS Engineering Interfaces **ccsei**, or with the MCM Engineering Interface **motei** engineers with tools to configure motor parameters and test motors without having to write code.

The main concepts described in section 2.4 and sub-sections apply to the ACI module as well and will not be repeated here.

2.5.1 Commands

The commands supported in the current version are listed below, they are grouped by topic.They will not be described in detail; see section 2.12 and following of the LCC Common Software User Manual [1] for further explanations.
Most of the command refer to the notion of set. A set is a group of motors that are intended to be controlled simultaneously, but which may perform independent actions. The commands that apply to a set also apply to a single motor, but the commands handling the sets (1 to 5).

<u>Set Handling</u>

1. **`CREASET`** Create a set of motors
2. **`ADDMOT`** Add a motor to a set of motors
3. **`REMMOT`** Remove a motor from a set
4. **`DELSET`** Delete a set
5. **`GMOTSET`** Get number and list of motors in a set or in all the sets

<u>Motor Handling</u>

6. **`INSTALL`** Install a motor in MCM
7. **`DEINST`** Deinstall a motor or all motors from MCM
8. **`MOTORS`** Get number and list of installed motors
9. **`ATTACH`** Attach a motor or the motor(s) of the set
10. **`DETACH`** Detach a motor or the motor(s) of a set
11. **`CLEAR`** Clear internal data structures (emergency case only)

<u>Commands</u>

12. **`CHKLOCK`** Check the status of the interlock line
13. **`CHKBRAK`** Check the brake status
14. **`CLAMP`** Clamp the brakes of the motors of the set
15. **`CONNECT`** Connect the power lines to the motors of the set
16. **`DISABLE`** Enable the axis of the motors of the set
17. **`DISCON`** Disconnect the power lines from the motors of the set
18. **`ENABLE`** Disable the axis of the motors of the set
19. **`GENSTAT`** Get the general status of the motors of the set

20. **GETOPMO** Get the actual operational mode of the motors of the set
21. **HWSTAT** Get the hardware status of the motors of the set
22. **INITDB** Restore configuration file into DB branch
23. **INITHW** Perform the HW initialization procedure on the motors of the set
24. **INITSAM** Start the status sampling for the motors of the set
25. **INITSW** Perform the SW initialization of the motors of the set
26. **LOADPAR** Load parameters to the board(s) controlling the motors of the set
27. **MOTMODE** Set the motion mode for the motors of the set
28. **MOTPAR** Retrieve the actual motion parameters of the motors of the set
29. **RESETSW** Reset the axis of the motors of the set
30. **RESETHW** Reset the boards controlling the motors of the set
31. **RETRPAR** Retrieve board parameters
32. **SECTION** Set the section for named positions handling
33. **SETADDR** Set the address of the user-defined data structure in the database
34. **SETLIM** Set the SW limits for the motors of the set
35. **SETOPMO** Set the operational mode for the motors of the set
36. **STATUS** Get the complete status of the motors of the set
37. **STOPSAM** Stop status sampling for the motors of the set
38. **UNCLAMP** Unclamp the brakes of the motors of the set
39. **WAITINI** Wait for HW initialization procedure completion
40. **WAITMOV** Wait for motion completion

Speed definition and Sequence management

41. **DEFSPD** Define the speed for a motion step
42. **DEFSTEP** Define a motion step
43. **REMSTEP** Remove a step in the motion sequence
44. **DELSEQ** Delete a motion sequence

Motion control

45. **MOVEABS** Perform absolute motion
46. **MOVEHOM** Perform motion to the Home position
47. **MOVEIND** Perform motion to the Index-Pulse of the encoder
48. **MOVELHW** Perform motion to the Lower HW limit
49. **MOVEREF** Perform motion to the Reference Switch
50. **MOVEREL** Perform relative motion
51. **MOVESEQ** Perform motion sequence
52. **MOVESPD** Perform motion by speed
53. **MOVEUHW** Perform motion to the Upper HW limit
54. **STOPMOT** Stop the motors of a set

Unit conversion

55. **DEFUNIT** Set the default units
56. **DIG2CUR** Conversion Dig To Current
57. **ENC2NAM** Conversion Enc To Name

58. **ENC2POS** Conversion Enc To Pos
59. **ENC2VEL** Conversion Enc ToVel
60. **NAM2ENC** Conversion NameTo Enc
61. **NAM2VEL** Conversion NameTo Vel
62. **POS2ENC** Conversion Pos To Enc
63. **REF2VEL** Conversion Ref ToVel
64. **VEL2ENC** Conversion Vel To Enc
65. **VEL2REF** Conversion Vel To Ref

Version

66. **VERSION** Get MCM API and ACI Versions

General commands

67. **INIT** See LCC Command INIT (not implemented)
68. **ONLINE** See LCC Command ONLINE (not implemented)
69. **SELFTST** See LCC Command SELFTST (not implemented)
70. **STANDBY** Set the motors of the set in stand-by mode (not implemented)
71. **STASIM** Start SIMULATION mode (not implemented)
72. **OFF** See LCC Command OFF (not implemented)
73. **STOP** See LCC Command STOP (not implemented)
74. **STOSIM** Stop SIMULATION mode (not implemented)
75. **EXIT** See LCC Command EXIT (terminates motServer)
76. **KILL** See LCC Command KILL (not implemented)

For detailed information on commands see section 3.2 and on-line help using **ccsei/motei**.

2.5.2 Input Syntax

All enumerated input parameters can be given either as numerical value (e.g. 4) or as the corresponding string (e.g. **Simulation**). The command interpreter is case sensitive.
The command **STATUS**, **MOTPAR**, **NAM2ENC** and **ENC2NAM** accept the enumerated parameter section, specifying the position section to search in for the position and next position names (if any). If no name is found the string "**None -1**" is returned. This parameter can be defaulted, the value "**No Section**" is then passed, the actual section is then used.

Example: The sign > prefixes a command, the actual position is assumed as '**blue**' in the **Maintenance** section.

```
>STATUS DC1,,PositionName
STATUS DC1 : blue Maintenance
>STATUS DC1,UserDefined,PositionName
STATUS DC1 : None -1
>STATUS DC1,Maintenance,PositionName
STATUS DC1 : blue Maintenance
```

The **STATUS** command accepts a parameter list as third parameter. If this list is empty, the complete status is returned. Otherwise, for each requested parameter (up to 10), the status value will be returned:

Example: The sign > prefixes a command

`>STATUS DC1`	returns the full status,
`>STATUS DC1,Observation`	returns the full status but the names are searched in the Observation section,
`>STATUS DC1,,MotionMode`	returns only the motion mode.

The **`WAITINI`** and **`WAITMOV`** commands are the only commands which do not return immediately the last reply. The first reply is the command acknowledge, the following replies are returned on function completion (in the meanwhile the command interpreter is available for other commands). One reply is returned for each specified motor. If an error occurred, an additional reply is returned containing the error message. The time-outs have to be given in **seconds**.

Example: The sign 12> prefixes the command #12

```
10>CREASET X,DC1,DC2                   create a set containing the motors DC1 and DC2
  10 REPLY CREASET
11>INITHW X                            perform HW initialization procedure,
  11 REPLY INITHW
12>WAITINI X,300                       wait max. 300s for completion
  12 REPLY WAITINI                     command accepted
13>STATUS DC1,,MotionMode              returns only the motion mode.
  13 REPLY STATUS DC1 : Speed          motor DC1 is in Speed mode
14>WAITINI DC1,5                       wait 5 seconds for completion for DC1 only
  14 REPLY WAITINI DC1 : ...,InitStatus H/W Initialization Running,...
  14 ERROR WAITINI 0x0CF30003.1 mot (motWaitInit.c)
      "motERR_MULTIPLE_WAIT : motor 'DC1', handle is 5023804 - Another
      application is already waiting for action completion" (Serious 40)
                                       command rejected
  12 REPLY WAITINI DC1 : ...,InitStatus Initialization Done,...
  12 REPLY WAITINI DC2 : ...,InitStatus Initialization Done,...
```

2.5.3 Output Syntax

The output syntax is built as follows:
For all the commands, but **`STATUS`**, **`MOTPAR`**, **`GENSTAT`** and **`HWSTAT`**, one reply is returned; otherwise one reply per motor is returned.

<reply>: command <motor>[,<motor>]^m

<motor>: motName : <param>[,<param>]^p

<param>: [parName] parValue [parUnit]

where p>=0 and m>=0.

For commands that query a single parameter (e.g. **`GETOPMO`**), the field **`parName`** is omitted. For position names, the field **`parValue`** takes the position name, the field **`parUnit`** takes the section; if no name can be found the fields **`parValue`** and **`parUnit`** take the value "**`None`**" and "**`-1`**", resp.

Example: The sign > prefixes a command

```
>CREASET X,DC1,DC2
>GETOPMO X
GETOPMO DC1 : Simulation,DC2 : Normal
>STATUS X,,OperationalMode MotionMode
```

```
STATUS DC1 : OperationalMode Simulation,MotionMode Direct
STATUS DC2 : OperationalMode Normal,MotionMode Direct
```

2.5.4 Output Format

The commands that query information from the motors accept an additional parameter (which may be ignored). This integer parameter takes the values 0 or 1 and is defaulted with 0. It influences the output format for enumerated parameters only:

0 : The returned values are given by their literals.

Example: "**Simulation**" for the operational mode **SIMULATION**. (Macro **motSIMULATION_STRING**).

1 : The returned valued are given by their numerical value.

Example: **4** for the operational mode **SIMULATION** (Macro **motSIMULATION**).

All literals for status parameter values are defined in **motDefines.h**.
The names of the status parameters are defined in **motciDefines.h**.

Example: The sign **>** prefixes a command

```
>SETOPMO DC1,Simulation
>SETOPMO DC2,5
>GETOPMO X
GETOPMO DC1 : Simulation,DC2 : Normal
>GETOPMO X,1
GETOPMO DC1 : 4,DC2 : 5
```

3 REFERENCE MANUAL

This chapter provides a detailed description of the **mot** API procedures and tools and **motci** ACI commands provided by these two modules.

The API functions and tools are described in UNIX man-page format and are organized in alphabetical order.
The ACI commands are described in the Command Definition Table of the process **`motServer`** (file **`motServer.cdt)`**.
Necessary include files are also listed.

3.1 Aplication Programmatic Interface

3.1.1 motAttach(3)

NAME

```
motAttach - attach a motor
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motAttach (
                         IN  dbSYMADDRESS    dbPath,
                         IN  motACCESS_MODE  mode,
                         OUT motHANDLE      *handle,
                         OUT ccsERROR       *error
                        )
```

DESCRIPTION

```
This routine attaches a motor to the user's application.
The motor is specified by its pathname <dbPath>, which is the absolute
path to the motor or its alias in the local database.
The parameter <mode> specifies the access rights to motor.
The returned <handle> identifies the channel to the motor and
is required for any further motor library function call.

Possible access modes are:
  - motREAD_ONLY : only read actions are possible
  - motEXCLUSIVE : all functions provided by the motor library
                   are allowed; a motor can be attached only
                   once in this mode
  - motEMERGENCY : for emergency case only

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if motor attached successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.2 motCheckBrake(3)

NAME

```
motCheckBrake - check the brake status of a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motCheckBrake (
                             IN  motHANDLE  handle[],
                             IN  vltINT32  *status,
                             OUT ccsERROR  *error
                            )
```

DESCRIPTION

```
The mot Server routine motCheckBrake() checks the status of the
brake for the set of motors, specified in the <handle> list.
It returns the values in the <status> array. The <status> array need
not provide space for a terminator entry.

This routine needs READ_ONLY access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable. The routine processes
all motors of the <handle> list, and does not stop execution when an
error occurs.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.3 motCheckInterlock(3)

NAME

```
motCheckInterlock - check interlock line status of a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motCheckInterlock (
                                 IN  motHANDLE  handle[],
                                 IN  vltINT32   *status,
                                 OUT ccsERROR   *error
                                )
```

DESCRIPTION

```
The mot Server routine motCheckInterlock() checks the status of the
interlock line for the set of motors, specified in the <handle> list.
It returns the values in the <status> array. The <status> array need
not provide space for a terminator entry.

This routine needs READ_ONLY access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable. The routine processes
all motors of the <handle> list, and does not stop execution when an
error occurs.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.4 motCheckMove(3)

NAME

```
motCheckMove - check validity of requested motion for a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motCheckMove (
                            IN  motHANDLE   handle[],
                            IN  motMOTION  *motionReq,
                            OUT ccsERROR   *error
                           )
```

DESCRIPTION

```
This routine check the validity of the requested motions for all
motors specified by the <handle> list.
The <motionReq> array must at first contain the motions for the first
motor in the <handle> list, with the 'lastStep' flag set in the last
motion for this motor. Then the motion entries for the second motor
in the <handle> list in the same way, and so on.
The motion mode for each motor must have been defined by a
previous motSetMotionMode() call.
Each movement represents one entry in <motionReq> array,
and is defined as follows :

<motionReq>->posType  - the position types are :
                     motABSOLUTE, motRELATIVE,
                     motLOWER_LIMIT, motUPPER_LIMIT,
                     motREFERENCE_SWITCH, motINDEX_PULSE,
                     motHOME_POSITION, motBY_SPEED,
                     motTWO_STEP_ABS, motTWO_STEP_REL
                     motHALF_TURN_ABS, motHALF_TURN_REL

<motionReq>->speed    - moving speed

For the motion mode motPOSITION_MODE & motTRACKING_MODE, the target
position must be provided, except for the position type motBY_SPEED.

<motionReq>->pos      - the target position

<motionReq>->lastStep - flag to indicate the last step of a sequence
                        for a motor (TRUE/FALSE)

The minimum access right is READ_ONLY. The validity of the motion
sequences is checked if the motor(s) is(are) not moving, but their
validity may change if a motion is performed after this call.
In order to avoid any dependency, it is recommended to start the
sequence from a known position.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The routine stops
execution when an error occurs or when a motion is illegal, and does
not process the remaining motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

SEE ALSO

```
motMove
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.5 motClampBrake(3)

NAME

```
motClampBrake - clamps the brakes of a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motClampBrake (
                             IN  motHANDLE  handle[],
                             OUT ccsERROR   *error
                            )
```

DESCRIPTION

```
The mot Server routine motClampBrake() clamps the brakes
of the set of motors, specified in the <handle> list.

This routine needs EXCLUSIVE access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description is returned in the <error> variable. The routine does not
break execution when an error occurs, but tries to process the
remaining motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if routine terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.6 motConnect(3)

NAME

```
motConnect - enable electrical connection of a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motConnect (
                          IN  motHANDLE  handle[],
                          OUT ccsERROR   *error
                         )
```

DESCRIPTION

```
The mot Server routine motConnect() enables the electrical connection
of the set of motors, specified in the <handle> list.

This routine needs EXCLUSIVE access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable. The routine stops
execution when an error occurs, nad does not process the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.7 motDetach(3)

NAME

```
motDetach - release the access to a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motDetach (
                         IN   motHANDLE  handle[],
                         OUT  ccsERROR   *error
                        )
```

DESCRIPTION

```
This routine releases the access to the motors, specified in the
<handle> list. If a motor is still moving, it is not detached and
an error is returned.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The processing is
stopped when the first error occurs.
```

FILES

```
None
```

ENVIRONMENT

```
None
```

RETURN VALUES

```
SUCCESS, if function terminated successfully
FAILURE, if any error occurred
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.8 motDigToCurrent(3)

NAME

```
motDigToCurrent - convert digital value to user defined current unit
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motDigToCurrent (
                               IN    motHANDLE  handle,
                               IN    vltINT32   currDig,
                               INOUT motMETRIC *currUser,
                               OUT   ccsERROR  *error
                              )
```

DESCRIPTION

```
The mot Server routine motDigToCurrent() converts a motor current
from the digital value <currDig> to the value <currUser> in user units.
The required unit has to be specified in <currUser>->unit
(for example "mA"), the value evaluated is returned by
<currUser>->value (double). The unit specified must be defined
in the local database.

The conversion is specified for the motor defined by <handle>.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.9 motDisable(3)

NAME

```
motDisable - disable motion control loop for a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motDisable (
                          IN  motHANDLE  handle[],
                          OUT ccsERROR   *error
                         )
```

DESCRIPTION

```
The mot Server routine motDisable() disables the motion control loop
for a set of motors, specified by the <handle> list. This function
is supported by motion controller boards only. If no controller is
configured an error is returned.

This routine needs EXCLUSIVE access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

SEE ALSO

```
Technical Description of Motion Controller
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.10 motDisconnect(3)

NAME

motDisconnect - disable electrical connection of a set of motors

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motDisconnect (
                              IN  motHANDLE  handle[],
                              OUT ccsERROR  *error
                             )
```

DESCRIPTION

The mot Server routine motDisconnect() disables the electrical connection of the set of motors, specified in the <handle> list.

This routine needs EXCLUSIVE access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and description is returned by the <error> variable. The routine stops execution when an error occurs, and does not process the remaining motors of the set.

FILES

none

ENVIRONMENT

none

RETURN VALUES

SUCCESS if function terminated successfully
FAILURE if anything went wrong

CAUTIONS

none

\- - - - - -
Last change: 30/09/98-10:36

3.1.11 motEnable(3)

NAME

```
motEnable - enable motion control loop for a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motEnable (
                         IN  motHANDLE handle[],
                         OUT ccsERROR  *error
                        )
```

DESCRIPTION

```
The mot Server routine motEnable enables the motion control loop for
the set of motors, specified by the <handle> list.
This function is supported by motion controller boards only.
If no controller is configured an error is returned.

This routine needs EXCLUSIVE access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

SEE ALSO

```
Technical Description of corresponding motion controller
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.12 motEncToName(3), motEncToIndex(3)

NAME

```
motEncToName - convert encoder ticks to named position
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motEncToName (
                            IN    motHANDLE   handle,
                            IN    vltINT32    posEnc,
                            INOUT motSECTION *section,
                            OUT   vltBYTES32  name,
                            OUT   ccsERROR   *error
                           )
```

DESCRIPTION

```
The mot Server routine motEncToName() tries to convert a position
<posEnc>, given in encoder ticks to a <name>, which corresponds to
a named position.
By <section> the type of position requested is specified.
Possible values are:
        - motOBSERVATION
        - motMAINTENACE
        - motUSER_DEFINED
        - motALL_SECTION

In case of motALL_SECTION, the associated named position is searched
in all sections, starting with OBSERVATION. The first matching named
position is returned. The name and section are set to the matching
values.
If no matching name is found,
  <name>    is set to empty string "",
  <section> is set to motNO_SECTION.

The database section SERVER_CONF:POSITIONS is queried for the named
positions.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.13 motEncToPos(3)

NAME

```
motEncToPos - convert encoder values to position in user units
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motEncToPos (
                           IN    motHANDLE  handle,
                           IN    vltINT32   posEnc,
                           INOUT motMETRIC *posUser,
                           OUT   ccsERROR  *error
                          )
```

DESCRIPTION

```
The mot Server routine motEncToPos converts the position from encoder
ticks <posEnc> to user units defined by <posUser>->unit
(for example "deg").
The value evaluated is returned by <posUser>->value (double).
The user unit specified must be defined in the local database.
The conversion is specific for the motor, defined by <handle>.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
The position must be absolute
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.14 motEncToVel(3)

NAME

```
motEncToVel - convert encoder speed values to velocity in user units
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motEncToVel (
                           IN    motHANDLE  handle,
                           IN    vltDOUBLE  velEnc,
                           INOUT motMETRIC *velUser,
                           OUT   ccsERROR  *error
                          )
```

DESCRIPTION

```
This routine motEncToVel() converts the velocity from encoder ticks
per time unit <velEnc> to user units specified by <velUser>->unit
(for example "deg/sec"). The value evaluated is returned by
<velUser>->value (double). The user unit specified must be defined
in the local database.
The conversion is specific for the motor, defined by <handle>.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.15 motGetOpMode(3)

NAME

```
motGetOpMode - get operational mode of a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motGetOpMode (
                            IN   motHANDLE  handle[],
                            OUT  motOPMODE  mode[],
                            OUT  ccsERROR   *error
                           )
```

DESCRIPTION

```
This routine returns the operational modes of all motors in the
specified <handle> list in the array <mode>.
Possible modes are:
            - motNOT_AVAILABLE
            - motFIXED_POSITION
            - motSIMULATION
            - motNORMAL
            - motHANDSET

Note: the <mode> array is not terminated by a special entry.

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
The <mode> array must be large enough to contain all the modes of the
requested motors.
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.16 motGetHandle(3), motGetSemIEV(3)

NAME

```
motGetSemIEV - Returns the semaphore for IEV Isr
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motGetSemIEV (
                             IN  motHANDLE  handle,
                             OUT SEM_ID    *semIEV,
                             OUT ccsERROR  *error
                            )
```

DESCRIPTION

```
The mot Server routine motGetSemIEV() retrieves the semaphore given
by the Interrupt Service Routine handling the external encoder value.
It returns FAILURE if the pointer to the semaphore is NULL.
The function waits forever until the semaphore has been created
by motInitSw().

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description is returned in the <error> variable.
```

FILES

none

ENVIRONMENT

none

RETURN VALUES

```
SUCCESS if routine terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

none

SEE ALSO

motWriteEncoderValue(3), motGetHandle(3)

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.17 motGetStatus(3)

NAME

motGetStatus - get status information of a set of motors

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motGetStatus (
                            IN    motHANDLE  handle[],
                            INOUT motSTATUS  status[],
                            OUT   ccsERROR  *error
                           )
```

DESCRIPTION

This routine returns status information for the specified set of motors in the <status> array with an entry for each motor specified by the <handle> list. Additionally the MOTOR_STATUS section of each motor database-point is updated accordingly.

The <status> data structure is build up in the following way:

parameter	meaning	possible values
opMode	operational mode	motNOT_AVAILABLE/motHANDSET
		motFIXED_POSITION/motNORMAL
		motSIMULATION
motionMode	actual motion mode	motDIRECT_MODE/motSPEED_MODE
		motTRACKING_MODE
		motPOSITION_MODE
motionStatus	motion status	motSTANDING/motMOVING
		motTIMEOUT/motABORTED
motionStep	number of motion	
initStatus	initialization status	motNOT_INT/motSW_INIT
		motHW_INIT_RUN
		motINIT_DONE
initStep	step of init sequence	
brakeClamped	brake status	(I/O signal)
axisEnabled	axis enabled flag	TRUE/FALSE
eventMask	bit mask to encode	motMOTION_END/motDRIVE_FAULT
	abnormal events	motEMERGENCY_STOP
		motOVER_TEMPERATURE
		motOVER_CURRENT/motON_LIMIT
		motPOSITIONING_ERROR
		motNORMAL_STOP
amplStatus	amplifier status	motNOT_AVAILABLE/OK/ERROR
powerStatus	electrical connection	motBOTH/motNEGATIVE
		motPOSITIVE/motDISCONNECTED
mconStatus	controller status	motNOT_AVAILABLE/OK/ERROR
inPos	in position flag	TRUE/FALSE
amplOnLimit	amplifier limit flag	motNOT_ON_LIMIT/motHW_UPPER
		motHW_LOWER
mconOnLimit	controller limit flag	motNOT_ON_LIMIT
		motSW_LOWER/motSW_UPPER
		motHW_LOWER/motHW_UPPER
		motHW_BOTH
interlock	interlock line status	(I/O signal)
speed	actual speed	
speedUnit	speed unit	for example "deg/sec"
		(type motUNIT)

current	actual motor current	
currentUnit	motor current unit	for example "mA" (type motUNIT)
posEnc	actual position in encoder ticks	
posUser	actual position in user units	
posUnit	position user unit	for example "deg" (type motUNIT)
posName	name of actual position	NULL, if none available
posSection	position section	motOBSERVATION/motMAINTENANCE motUSER_DEFINED/motNO_SECTION
nextPosEnc	next position in encoder ticks	
nextPosUser	actual position in user units (posUnit)	
nextPosName	name of next position	NULL, if none available
nextPosSection	next position section	motOBSERVATION/motMAINTENANCE motUSER_DEFINED/motNO_SECTION
followingErr	actual following error in user units (posUnit)	
startTime	start time of movement in UTC	
endTime	end time of movement in UTC	

If anything goes wrong FAILURE is returned and the error reason and description are returned by the <error> variable. The routine stops execution when an error occurs, and does not process the remaining motors of the set.

RETURN VALUES

SUCCESS if function terminated successfully
FAILURE if anything went wrong

CAUTIONS

none

\- - - - - -
Last change: 30/09/98-10:36

3.1.18 motInitHw(3)

NAME

motInitHw - initialize board hardware for specified motors

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motInitHw (
                         IN  motHANDLE   handle[],
                         IN  vltLOGICAL  mandatory,
                         OUT ccsERROR   *error
                        )
```

DESCRIPTION

```
The mot Server routine motInitHw initializes the board hardware of
the boards controlling the motors specified in the <handle> list.
The <mandatory> flag specifies whether the initialization has to be
performed in any case (TRUE) or only, if not done before (FALSE).

If any user functions are used for HW initialization, pointers
to user defined data structures to be passed to the functions are
located in the Database :SERVER:INIT:INIT:action[].userParList.
The following parameters will be passed to the user function:
       - pointer to user defined data structure
       - error structure address

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
motors of the set.
```

FILES

none

ENVIRONMENT

none

RETURN VALUES

SUCCESS if function terminated successfully
FAILURE if anything went wrong

CAUTIONS

none

\- - - - - -
Last change: 30/09/98-10:36

3.1.19 motInitSampling(3)

NAME

motInitSampling - initialize status sampling for a set of motors

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motInitSampling (
                               IN  motHANDLE  handle[],
                               IN  vltINT32   period,
                               IN  motSECTION section,
                               OUT ccsERROR   *error
                              )
```

DESCRIPTION

```
The mot Server routine motInitSampling starts the status sampling
for the motors, specified in the <handle> list. The sample data is
stored in intervals of <period> in the local database general status
section. For each motor one sampling task is spawned.
The named positions are searched for in the given <section>.
Depending on the <period>, the effective sampling period is:
range [-100 ; -2] : the period is 1/|<period>| in seconds
value  -1         : as fast as possible
value   0         : set to the DB value SERVER:CONF.motor(0,timemon).
range [+1 -> )    : the period is <period> in seconds
The minimum <period> is -100, any setting below is forced to 100Hz.

Since the time to retrieve the boards' status is approx. 10ms, and

This routine needs READ_ONLY access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable.
```

FILES

none

ENVIRONMENT

none

RETURN VALUES

SUCCESS if function terminated successfully
FAILURE if anything went wrong

CAUTIONS

none

\- - - - - -
Last change: 30/09/98-10:36

3.1.20 motInitSw(3)

NAME

```
motInitSw - initialize board software for specified motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motInitSw (
                         IN  motHANDLE   *handle,
                         IN  vltLOGICAL  mandatory,
                         OUT ccsERROR    *error
                        )
```

DESCRIPTION

```
The mot Server routine motInitSw initializes the board software of
the boards controlling the motors, specified in the <handle> list.
The <mandatory> flag specifies whether the initialisation has to be
performed in any case (TRUE) or only, if not done before (FALSE).

If anything goes wrong FAILURE is returned and the error reason and
description are returned in the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.21 motLoadPar(3)

NAME

motLoadPar - load parameters to a board, controlling a motor

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motLoadPar (
                          IN    motHANDLE      handle,
                          IN    motBOARD_TYPE  board,
                          INOUT vltUINT32     *number,
                          IN    motPARAMETER  *parBuf,
                          OUT   ccsERROR       *error
                         )
```

DESCRIPTION

The mot Server routine motLoadPar() loads board specific parameters for the motor, specified through its <handle> to the <board> of the specified type.
Possible board types are motAMPLIFIER/motCONTROLLER.
The number of parameters to be loaded is passed by <number>, parameter names and values are defined by the <parBuf> array (<parBuf>->parName/<parBuf>->parValue).
After termination of this function <number> contains the number of parameters successfully loaded.

This routine needs EXCLUSIVE access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and description are returned by the <error> variable. The routine stops execution when an error occurs, and does not process the remaining parameters.

FILES

none

ENVIRONMENT

none

RETURN VALUES

SUCCESS if function terminated successfully
FAILURE if anything went wrong

CAUTIONS

This routine is for ONE motor only, not for a set of motors. It is intended to be used for maintenance purposes only.

\- - - - - -
Last change: 30/09/98-10:36

3.1.22 motMotors(3)

NAME

```
motMotors - Returns the number and list of installed motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motMotors ( int *num, vltBYTES20 list[] )
```

DESCRIPTION

```
This function returns in the string pointed to by <list> the list of
the installed motors using the DB aliases.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

CAUTIONS

```
If the list is to be returned in the string <list>, enough space MUST
have been allocated (up to 20 bytes per motor).
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.23 motMove(3)

NAME

```
motMove - start movement of a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motMove (
                       IN  motHANDLE  handle[],
                       IN  motMOTION *motionReq,
                       OUT ccsERROR  *error
                      )
```

DESCRIPTION

```
This routine initiates motions for all motors specified by the
<handle> list.
For each motor a sequence of motions can be defined by <motionReq>.
For each movement one entry in <motionReq> must contain:

<motSeq>->posType - the position type
                    (motABSOLUTE/motRELATIVE/motLOWER_LIMIT/
                     motUPPER_LIMIT/motREFERENCE_SWITCH/motINDEX_PULSE/
                     motHOME_POSITION/motBY_SPEED/
                     motTWO_STEP_ABS/motTWO_STEP_REL/
                     motHALF_TURN_ABS/motHALF_TURN_REL)
<motSeq>->speed    - moving speed
<motSeq>->pos      - the position to move to (motABSOLUTE/motRELATIVE)
<motSeq>->lastStep - flag to indicate the last step of a sequence
                     for a motor  (TRUE/FALSE)

The <motionReq> array must at first contain the motions for the first
motor in the <handle> list, with the 'lastStep' flag set in the last
motion for this motor. Then the motion entries for the second motor
in the <handle> list in the same way, and so on.
The motion mode for each motor must have been defined by a previous
motSetMotionMode() call.

If the motor is electrically disconnected, it will be automatically
connected.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
            none



- - - - - -
Last change:  30/09/98-10:36
```

3.1.24 motNameToEnc(3)

NAME

```
motNameToEnc - convert named position to encoder ticks
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motNameToEnc (
                            IN  motHANDLE   handle,
                            IN  vltBYTES32  name,
                            IN  motSECTION  section,
                            OUT vltINT32   *posEnc,
                            OUT ccsERROR   *error
                           )
```

DESCRIPTION

```
The mot Server routine motNameToEnc() converts a named position,
specified by its <name> and <section> to the corresponding encoder
ticks and returns its value by <posEnc>.
Possible values for <section> are:   - motOBSERVATION
                                     - motMAINTENANCE
                                     - motUSER_DEFINED

The routine works specific for the motor <handle>.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.25 motNameToVel(3)

NAME

```
motNameToEnc - convert named position to encoder ticks
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motNameToEnc (
                            IN  motHANDLE   handle,
                            IN  vltBYTES32  name,
                            IN  motSECTION  section,
                            OUT vltINT32   *posEnc,
                            OUT ccsERROR   *error
                           )
```

DESCRIPTION

```
The mot Server routine motNameToEnc() converts a named position,
specified by its <name> and <section> to the corresponding encoder
ticks and returns its value by <posEnc>.
Possible values for <section> are:   - motOBSERVATION
                                     - motMAINTENANCE
                                     - motUSER_DEFINED

The routine works specific for the motor <handle>.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.26 motPosToEnc(3)

NAME

motPosToEnc - convert position to encoder values

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motPosToEnc (
                           IN  motHANDLE  handle,
                           IN  motMETRIC *posUser,
                           OUT vltINT32  *posEnc,
                           OUT ccsERROR  *error
                          )
```

DESCRIPTION

The mot Server routine motPosToEnc() converts the position in user units <posUser> (for example "deg") to an encoder specific value and returns it by <posEnc>. The user unit specified must be defined in the local database.
The conversion is specific for the motor defined by <handle>.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and description are returned by the <error> variable.

FILES

none

ENVIRONMENT

none

RETURN VALUES

SUCCESS if function terminated successfully
FAILURE if anything went wrong

CAUTIONS

The position must be absolute

\- - - - - -
Last change: 30/09/98-10:36

3.1.27 motRefToVel(3)

NAME

```
motRefToVel - convert ampl velocity reference to velocity in user units
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motRefToVel (
                           IN  motHANDLE  handle,
                           IN  vltINT32   velRef,
                           OUT motMETRIC *velUser,
                           OUT ccsERROR  *error
                          )
```

DESCRIPTION

```
The mot Server routine motRefToVel() converts the velocity from
amplifier velocity reference <velRef> to user units <velUser>
(for example "deg/sec").
The user unit specified must be defined in the local database.
The conversion is specific for the motor, defined by <handle>.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable.
```

FILES

none

ENVIRONMENT

none

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

none

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.28 motRealAxis(3)

See motResetSw(3).

```
- - - - - - -
Last change:  30/09/98-10:36
```

3.1.29 motResetBoards(3)

See motResetHw(3).

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.30 motRetrievePar(3)

NAME

```
motRetrievePar - retrieve parameters from a board, controlling a motor
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motRetrievePar (
                              IN    motHANDLE      handle,
                              IN    motBOARD_TYPE  board,
                              INOUT vltUINT32     *number,
                              INOUT motPARAMETER  *parBuf,
                              OUT   ccsERROR       *error
                             )
```

DESCRIPTION

```
The mot Server routine motRetrievePar retrieves board specific
parameters of the motor, specified by <handle> from the <board> of the
specified type.
Possible board types are motCONTROLLER and motAMPLIFIER.
The number of parameters to be retrieved is specified by <number>.
Each parameter is specified by its name in the corresponding <parBuf>
entry (<parBuf->parName), whereby the parameter value will be returned
by <parBuf>->parValue.
After termination of this function <number> contains the number of
parameters successfully read.

This routine needs EXCLUSIVE access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
parameters.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
This routine is for ONE motor only, not for a set of motors. It is
intended to be used for maintenance purposes only.
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.31 motSetDefaultUnits(3)

NAME

```
motSetDefaultUnits - set default user units for a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motSetDefaultUnits (
                                  IN  motHANDLE        *handle,
                                  IN  motDEFAULT_UNITS *units,
                                  OUT ccsERROR         *error
                                 )
```

DESCRIPTION

```
The mot Server routine motSetDefaultUnits() defines the default <units>
for position, speed and electric current for the set of motors
specified in the <handle> list. The default units are stored in the
local database and are used in the motor status, where parameters are
noted in user units.

This routine needs READ_ONLY access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.32 motSetMotionMode(3)

NAME

```
motSetMotionMode - set motion mode for a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motSetMotionMode (
                                IN  motHANDLE      handle[],
                                IN  motMOTIONMODE  mode[],
                                OUT ccsERROR      *error
                               )
```

DESCRIPTION

```
This routine sets the motion mode of each motor specified by the
handle list <handle>. The mode array need not be terminated.
Supported motion modes are:
  - DIRECT   : control through amplifier only
  - POSITION : move to absolute or relative position
  - TRACKING : follow a sequence of positions
  - SPEED    : drive with a constant speed, no target position

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
motors of the list.
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.33 motSetOpMode(3)

NAME

motSetOpMode - select the operational mode of a set of motors

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motSetOpMode (
                            IN  const motHANDLE  handle[],
                            IN  const motOPMODE  opMode[],
                            OUT       ccsERROR  *error
                           )
```

DESCRIPTION

```
This routine sets all motors in the specified <handle> list to the
operational mode as specified in the corresponding entry of the
<opMode> array.
Possible operational modes are: - motNOT_AVAILABLE
                                - motFIXED_POSITION
                                - motSIMULATION
                                - motNORMAL
                                - motHANDSET

If a required mode is equal to the current mode, no error is returned.
The <opMode> array need not have a terminating entry, unlike the
<handle> array.

If the operational mode is set to NORMAL or HANDSET the motor is
automatically electrically connected; in any other case the motor
is disconnected.

If a motion controller is available the control loop will be
automatically disabled.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The routine stops
execution when an error occurs, and does not process the remaining
motors of the set.
```

FILES

none

ENVIRONMENT

none

RETURN VALUES

SUCCESS if function terminated successfully
FAILURE if anything went wrong

CAUTIONS

This routine needs EXCLUSIVE access right to the motors.

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.34 motGetParListAddress(3), motSetParListAddress(3)

NAME

```
motSetParListAddress, motGetParListAddress - Write / read the address
   of the user data structure to the Database
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motSetParListAddress (
                                   IN  motHANDLE   handle,
                                   IN  motUSER_FCT fctType,
                                   IN  vltINT32    fctOccurence,
                                   IN  vltBYTES32  fctName,
                                   IN  vltUINT32   parList,
                                   OUT ccsERROR   *error
                                  )

ccsCOMPL_STAT motGetParListAddress (
                                   IN  motHANDLE   handle,
                                   IN  motUSER_FCT fctType,
                                   IN  vltINT32    fctOccurence,
                                   IN  vltBYTES32  fctName,
                                   IN  vltUINT32  *parList,
                                   OUT ccsERROR   *error
                                  )
```

DESCRIPTION

```
The two routines motSetParListAddress() & motGetParListAddress()
writes / reads in the Database the address <parList> of the data
structure to be used by the specified user function <fctName>
for the given motor <handle>.

According to the function type <fctType>, the routine will determine
where to access the <parList> value in the Database.
- motUSER_INIT :      :SERVER:INIT:INIT.action[].userParList
- motUSER_CONV :      :SERVER:UNITS:UNITS.unitConversion[].userParList

The parameter <fctOccurence> defines the accessing mode of the address:
- 0 : all occurences of the function in the table are updated
      (only for motSetParListAddress())
- 1 : the first occurence only
- 2 : the second occurence only
- etc ...

This routine needs EXCLUSIVE access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

\- - - - - -

Last change: 30/09/98-10:36

3.1.35 motSetSwLimits(3)

NAME

```
motSetSwLimits - set SW limits of a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motSetSwLimits (
                              IN  const motHANDLE    handle[],
                              IN        motPOSITION *positions,
                              IN        motPOS_TYPE *posTypes,
                              IN        motLIMITS   *limitFlags,
                              OUT       ccsERROR    *error
                             )
```

DESCRIPTION

```
The mot Server routine motSetSwLimits() sets the software limits
of the set of motors, specified in the <handle> list.

This routine needs EXCLUSIVE access right to the motors.

For each motor, an entry in the array <limitFlags> has to be provided
specifying the limits to be set.
The possible values are : motSW_UPPER / motSW_LOWER / motSW_BOTH.
Dependent on this flag, the limits have to be defined in the array
<positions>. If both limits have to be set, the lower limit has to be
defined first in the array <position>.

The <posTypes> array must built as <positions> and defines the setting
type of the limit, i.e.:
  - motABSOLUTE         : limit is given as an absolute position
  - motRELATIVE         : limit is given relative to actual position
  - motLOWER_LIMIT      : limit is given relative to lower HW limit
  - motUPPER_LIMIT      : limit is given relative to upper HW limit
  - motREFERENCE_SWITCH : limit is given relative to the ref switch
  - motINDEX_PULSE      : limit is given relative to the index pulse
  - motHOME_POSITION    : limit is given relative to the home position

If anything goes wrong FAILURE is returned and the error reason and
description is returned by the <error> variable. The routine stops
execution when an error occurs, nad does not process the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.36 motStop(3)

NAME

```
motStop - stop the motion of a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motStop (
                       IN  motHANDLE *handle,
                       OUT ccsERROR  *error
                      )
```

DESCRIPTION

```
The mot Server routine motStop() stops all motors specified in the
<handle> list. The Motion Monitor is informed to stop monitoring these
motors. If a motor has already stopped no error is returned.

This routine needs EXCLUSIVE access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The routine does not
stop execution when an error occurs, and processes the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.37 motStopSampling(3)

NAME

motStopSampling - stop status sampling for a set of motors

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motStopSampling (
                               IN  motHANDLE  handle[],
                               OUT ccsERROR  *error
                              )
```

DESCRIPTION

The mot Server routine motStopSampling() stops the status sampling for all motors specified by the <handle> list.

Note: This routine stops when an error occurs, and does not process the remaining motors.

This routine needs READ_ONLY access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and description are returned by the <error> variable.

FILES

none

ENVIRONMENT

none

RETURN VALUES

SUCCESS if function terminated successfully
FAILURE if anything went wrong

CAUTIONS

none

- - - - - -
Last change: 30/09/98-10:36

3.1.38 motUnclampBrake(3)

NAME

```
motUnclampBrake - unclamps the brakes of a set of motors
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motUnclampBrake (
                               IN  motHANDLE  handle[],
                               OUT ccsERROR  *error
                              )
```

DESCRIPTION

```
The mot Server routine motUnclampBrake() unclamps the brakes
of all motors specified by the <handle> list.

This routine needs EXCLUSIVE access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable. The routine does not
stop execution when an error occurs, but tries to process the remaining
motors of the set.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.39 motVelToEnc(3)

NAME

```
motVelToEnc - convert velocity in user units to encoder ticks
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motVelToEnc (
                           IN  motHANDLE  handle,
                           IN  motMETRIC *velUser,
                           OUT vltDOUBLE *velEnc,
                           OUT ccsERROR  *error
                          )
```

DESCRIPTION

```
The mot Server routine motVelToEnc() converts the velocity in user
units <velUser> (for example "deg/sec") to an encoder specific value
and returns it by <velEnc>. The user unit specified must be defined
in the local database.
The conversion is specific for the motor defined by <handle>.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description are returned by the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.40 motVelToRef(3)

NAME

motVelToRef - convert velocity in user units to ampl velocity reference

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motVelToRef (
                           IN  motHANDLE  handle,
                           IN  motMETRIC *velUser,
                           OUT vltINT32  *velRef,
                           OUT ccsERROR  *error
                          )
```

DESCRIPTION

The mot Server routine motVelToRef() converts the velocity in user units <velUser> (for example "deg/sec") to an amplifier velocity reference value and returns it by <velRef>. The user unit specified must be defined in the local database.

The conversion is specific for the motor defined by <handle>.

This routine needs READ_ONLY access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and description are returned by the <error> variable.

FILES

none

ENVIRONMENT

none

RETURN VALUES

SUCCESS if function terminated successfully
FAILURE if anything went wrong

CAUTIONS

none

\- - - - - -
Last change: 30/09/98-10:36

3.1.41 motWaitInit(3)

NAME

motWaitInit - wait for HW initialization completion of a set of motors

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motWaitInit (
                           IN  motHANDLE  handle[],
                           IN  vltINT32   timeout,
                           IN  motSTATUS *statusBuffer,
                           OUT ccsERROR  *error
                          )
```

DESCRIPTION

The routine motWaitInit is used to wait for the initialization
completion of all motors specified by the <handle> list.
The routine waits until all initializations have been completed
within <timeout> milliseconds, passed this time it returns FAILURE.
The routine returns FAILURE as soon an initialization failed,
the remaining motors of the list are ignored.
The current status of all motors is returned in the
<statusBuffer> array, one entry per motor in the same order as the
handle list, but without a termination entry.

Multiple use of this function by several applications for the same
motor is not possible.

This routine needs READ_ONLY access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and
description are returned in the <error> stack. The routine stops
execution when an error occurs, and does not process the remaining
motors of the list.

FILES

none

ENVIRONMENT

none

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if a time-out occurred
           an initialization failed
           an other task is already waiting for one motor
           an internal error occurred while initiating the wait
           or reading the motor status.
```

CAUTIONS

none

SEE ALSO

motGetStatus(), motWaitMove()

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.42 motWaitMove(3)

NAME

motWaitMove - wait for motion completion of a set of motors

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motWaitMove (
                           IN  motHANDLE  handle[],
                           IN  vltINT32   timeout,
                           IN  motSTATUS *motStatus,
                           OUT ccsERROR  *error
                          )
```

DESCRIPTION

The routine motWaitMove is used to wait for the motion completion of all motors specified by the <handle> list.
The routine waits until all motion sequences have been completed within <timeout> milliseconds, passed this time it returns FAILURE.
The routine returns FAILURE as soon a motion failed, the remaining motors of the list are ignored.
The current status of all motors is returned in the <motStatus> array, one entry per motor in the same order as the handle list, but without a termination entry.

Multiple use of this function by several applications for the same motor is not possible.

This routine needs READ_ONLY access right to the motors.

If anything goes wrong FAILURE is returned and the error reason and description are returned in the <error> stack. The routine stops execution when an error occurs, and does not process the remaining motors of the list.

FILES

none

ENVIRONMENT

none

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if a time-out occurred
           a motion failed
           an other task is already waiting for one motor
           an internal error occurred while initiating the wait
           or reading the motor status.
```

CAUTIONS

none

SEE ALSO

motGetStatus(), motWaitInit()

```
- - - - - -
Last change:  30/09/98-10:36
```

3.1.43 motWriteEncoderValue(3)

NAME

```
motWriteEncoderValue - Write encoder value to motion controller
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motWriteEncoderValue (
                                    IN  const motHANDLE  handle,
                                    IN        vltINT32   encoderValue,
                                    OUT       ccsERROR  *error
                                   )
```

DESCRIPTION

```
The mot Server routine motWriteEncoderValue() writes the
<encoderValue> to the Motion Controller board.

This routine needs EXCLUSIVE access right to the motor.

If anything goes wrong FAILURE is returned and the error reason and
description is returned in the <error> variable.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if routine terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
none
```

SEE ALSO

```
motGetSemIEV(3), motGetHandle(3)
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.2 Application Command Interface

Considered the size of the command definition table, the reader is invited to refer directly to the file **$VLTROOT/CDT/motServer.cdt.**
All the parameters and returned values are either character strings or binary formatted numbers. Help texts, command and parameter description are available on-line using **lccei**.

```
============================================================================
PUBLIC_COMMANDS

============================================================================
 Commands to create, modify and delete motor sets
============================================================================
CREASET        ADDMOT         REMMOT         DELSET         GMOTSET

============================================================================
 Commands to get/release access to a motor or a set of motors
============================================================================
INSTALL        MOTORS         DEINST
ATTACH         DETACH         CLEAR

============================================================================
 Control Commands
============================================================================
CHKLOCK        CHKBRAK        CLAMP          CONNECT        DISABLE
DISCON         ENABLE         GENSTAT        GETOPMO        HWSTAT
INITDB         INITHW         INITSAM        INITSW         LOADPAR
MOTMODE        MOTPAR         RESETAX        RESETBO        RETRPAR
SETADDR        SETLIM         SETOPMO        STATUS         STOPSAM
UNCLAMP        WAITMOV        WAITINI

============================================================================
 Commands to define/modify motion sequences
============================================================================
DEFSPD         DEFSTEP        REMSTEP        DELSEQ

============================================================================
 Commands to start/stop motions
============================================================================
MOVEABS        MOVEREL        MOVEUHW        MOVELHW        MOVEREF
MOVEIND        MOVEHOM        MOVESPD        MOVESEQ        STOPMOT

============================================================================
 Commands for unit conversion
============================================================================
DEFUNIT        DIG2CUR        ENC2NAM        ENC2POS        ENC2VEL
NAM2ENC        POS2ENC        REF2VEL        VEL2ENC        VEL2REF
VEL2NAM        SECTION

============================================================================
 general ...
============================================================================
VERSION

============================================================================
 lcc standard commands
============================================================================
```

```
INIT           STANDBY        ONLINE         OFF            STOP
EXIT           STASIM         STOSIM         SELFTST        KILL

============================================================================

MAINTENANCE_COMMANDS

============================================================================

TEST_COMMANDS

============================================================================
```

3.3 Tools

3.3.1 motClear(1)

NAME

```
motClear -  clear motor descriptor
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motClear (
                        IN dbSYMADDRESS dbPath
                       )
```

DESCRIPTION

```
The mot server routine motClear is used to reinitialize the descriptor
of the motor defined by <dbPath> (absolute symbolic path or alias)
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if an error occurred
```

CAUTIONS

```
No checks with regard to access rights etc. will be done. Any user
attached to the motor will be detached immediately.
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.3.2 motInitDb(1)

NAME

```
motInitDb - initialize database
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motInitDb ( char *motor, char *file, ccsERROR *error )
```

DESCRIPTION

```
This function restores  the MCM database for the DB branch
@<lcuEnv><alias><motor>, taking the values from the '.dbcfg' <file>.

<motor>  IN   name of the camera
<file>   IN   absolute pathname of the '.dbcfg' file
<error>  OUT  error structure
```

RETURN VALUES

```
SUCCESS if everything ok
FAILURE if anything went wrong
```

CAUTIONS

```
It is assumed that the FS containing the file to restore is NFS mounted.
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.3.3 motInstall(1)/motDeinstall(1)

NAME

```
motInstall - install a motor in MCM
```

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motInstall ( IN char *dbPath, IN ccsERROR *error )
```

DESCRIPTION

```
This routine initializes the motor control module "mot" (first call),
and the motor designated by the argument <dbPath>,
database path to the motor point (symbolic or alias).

The routine checks whether the Local Database exists and the motor
section is accessible and then initializes all sub-modules: API and SDL

If anything goes wrong FAILURE is returned and the error reason and
description are logged.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
SUCCESS if function terminated successfully
FAILURE if anything went wrong
```

CAUTIONS

```
This routine must be called once for each motor at system start-up.
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.3.4 motPrintMotors(1)

NAME

motPrintMotors - Display the list of installed motors

SYNOPSIS

```
#include "mot.h"

ccsCOMPL_STAT motPrintMotors ( void )
```

DESCRIPTION

This function displays the list of installed motors on the LCU console.

FILES

none

ENVIRONMENT

none

SUCCESS

FAILURE if ccsInit, ccsExit or dbAliasToName fail.

CAUTIONS

none

\- - - - - -
Last change: 30/09/98-10:36

3.3.5 motVersion(1)

NAME

```
motVersion - Print Motor Library Module Version
```

SYNOPSIS

```
void motVersion ( vltBYTES80 version )
```

DESCRIPTION

```
Extracts the version number of the installed Motor Control Module.
If <version> is NULL, the line is printed to the console.
```

FILES

```
none
```

ENVIRONMENT

```
none
```

RETURN VALUES

```
none
```

CAUTIONS

```
none
```

```
- - - - - -
Last change:  30/09/98-10:36
```

3.4 Include files

In order to get the necessary MCM and LCC data type definitions and function prototypes, the file **mot.h** must be included by any application source file referencing API functions and/or data structures. The file **motDefines.h** contains all the macros used in MCM. The file **motPublic.h** contains additional data type and structure definitions mapping database data structures. This file is intended to be included by SDL modules only, thus it should not be used at API level.

3.4.1 mot.h

```
/*******************************************************************************
* E.S.O. - VLT project
*
* "@(#) $Id: mot.h,v 2.7 1998/09/18 14:03:28 vltsccm Exp $"
*
* who       when      what
* --------  --------  ----------------------------------------------
* CAM GmbH  --/05/94  Created  for Release 1.1
* P.Duhoux  --/07/94  Modified for Release 1.2
* P.Duhoux  09/09/94  Renamed motVERSION_NUMBER to motVERSION
*                     and changed its definition
* P.Duhoux  12/05/95  Modified interface to motPrintMotors ( void )
*                     New function motMotors()
* P.Duhoux  09/06/95  Modified for CMM Archive
* P.Duhoux  07/07/95  Removed all literal definitions and
*                     include motDefines.h instead
* P.Duhoux  25/07/95  Added new function motCheckBrake()
* P.Duhoux  26/03/96  Added motStatus
*                      overCurrent,overTemperature
*                      onLimit,emergencyStop,driveFault,posError
*                      posIndex,nextPosIndex
* P.Duhoux  29/03/96  Added new functions:
*                      motGetSemIEV() & motWriteEncoderValue()
* P.Duhoux  12/09/96  Added new function motEncToIndex(), motGetHandle()
* P.Duhoux  13/09/96  Added new function motSetSection()
* P.Duhoux  10/11/96  Added new function motGetDbStatus()
* P.Duhoux  12/11/96  Moved definitions from motPublic.h for
*                      motCONFIG data structure,
*                     Added new function motGetDbConfig(), motFreeConfig()
* P.Duhoux  19/12/96  Added 'par' in motMOTION for INIT HW (initCode/waitDelay)
* P.Duhoux  14/01/97  Added functions motResetSw/motResetHw equivalent to
*                     motResetAxis/motResetBoards (for cosmetics)
*/

#ifndef MOT_H
#define MOT_H

/*******************************************************************************
* mot.h - motor control module API definition file
*-------------------------------------------------------------------------------
*/
#ifdef __cplusplus
extern "C" {
#endif

/*
```

```
 ******************************************************************************
 * Header Files
 ******************************************************************************
 */

/*
 * LCC include files
 */
#include "CCS.h"
#include "cai.h"

/*
 * MCM include files
 */
#include "motDefines.h"
#include "motErrors.h"

/*
 ******************************************************************************
 * Literals
 ******************************************************************************
 */

/* MOT module name */
#define motMODULE               "mot"
#define motMODULE_NAME          "Motor Control Module API"

/* literal for motor library version */
#define motVERSION              "$Revision: 2.7 $"
#define motVERSION_DATE         "SEP98"

/*
 ******************************************************************************
 * Data Types
 ******************************************************************************
 */

/*
 * definition of motor handle
 */
typedef vltUINT32 motHANDLE;

/* data type for access modes */
typedef vltINT32 motACCESS_MODE;

/* data type for operational modes */
typedef vltINT32 motOPMODE;

/* data type for units */
typedef vltBYTES8 motUNIT;

/* data type for axis types */
typedef vltINT32 motAXIS_TYPE;

/* data type for encoder types */
typedef vltINT32 motENC_TYPE;

/* data type for motor types */
```

```
typedef vltINT32 motMOTOR_TYPE;

/* data type for encoder coding types */
typedef vltINT32 motENC_CODE;

/* data type for abnormal event types */
typedef vltINT32 motEVENT_TYPE;

/* data type for digital IO signal types */
typedef vltINT32 motSIGNAL_TYPE;

/* data type for position/speed definition types */
typedef vltINT32 motSPEC_TYPE;

/* data type for section types */
typedef vltINT32 motSECTION;

/* data type for position types */
typedef vltINT32 motPOS_TYPE;

/* data type for motion modes */
typedef vltINT32 motMOTIONMODE;

/* data type for electrical connection types for amplifier boards */
typedef vltINT32 motCONNECTION;

/* data type for board types */
typedef vltINT32 motBOARD_TYPE;

/* data type for limit types */
typedef vltINT32 motLIMITS;

/* data type for motion status types */
typedef vltINT32 motMOTION_STATUS;

/* data type for initialization status types */
typedef vltINT32 motINIT_STATUS;

/* data structure for parameters */
typedef struct
    {
    vltBYTES20      parName;
    vltINT32        parValue;
    } motPARAMETER;

/* data structure for default units */
typedef struct
    {
    motUNIT position;
    motUNIT speed;
    motUNIT current;
    } motDEFAULT_UNITS;

/* data structure for metric number */
typedef struct
    {
    vltDOUBLE value;
    motUNIT   unit;
```

```
    } motMETRIC;

/* data structure for speed definition */
typedef struct
    {
    motSPEC_TYPE how;         /* speed definition mode                         */
    motMETRIC    number;      /* speed literal (value and unit)                */
    vltBYTES32   name;      /* speed name                                    */
    vltUINT32    index;       /* index in speed name table                     */
    vltUINT32    pollInt;     /* polling interval in millisec                  */
    vltUINT32    endPollInt;  /* end-phase polling interval in millisec        */
    vltUINT32    inPosTime;   /* needed in-position count for motion           */
                              /* complete condition in ms                      */
    } motSPEED;

/* data structure for position name */
typedef struct
    {
    vltBYTES32 posName;
    motSECTION section;
    } motPOS_NAME;

/* data structure for position definition */
typedef struct
    {
    motSPEC_TYPE how;         /* position definition mode          */
    motMETRIC    number;      /* position literal (value and unit) */
    motPOS_NAME  name;       /* named position                    */
    vltUINT32    index;       /* name index                        */
    } motPOSITION;

/* data structure for motion definition */
typedef struct
    {
    motPOS_TYPE   posType;/* position type                          */
    motPOSITION   pos;/* position                               */
    motSPEED      speed;/* speed                                  */
    vltINT32      par;         /* initCode/waitDelay (INIT only)         */
    vltLOGICAL    lastStep;/* flag for "last step in motion sequence" */
    } motMOTION;

typedef struct
    {
    motMOTOR_TYPE  type;      /* DC | STEPPER                      */
    vltINT32       timeout;   /* motion timeout in milliseconds    */
    vltINT32       timelim;   /* timeout for motion out of limit   */
    vltINT32       timemon;   /* automatic monitoring period       */
    motOPMODE      opMode;    /* operational mode on first attach */
    vltINT32       posEnc;    /* Fixed Position definition         */
    vltDOUBLE      posUser;
    vltBYTES8      posUnit;
    vltBYTES32     posName;
    vltINT32       posSection;
    vltINT32       chkHwLimit;/* Check flag for HW limits          */
    } motMOTOR_DEF;

/* database map for board configuration */
typedef struct
```

```
    {
    motBOARD_TYPE type;
    vltINT32      available;
    vltBYTES8     prefix;
    } motBOARD_CONF;

typedef struct
    {
    vltUINT32       name;
    motEVENT_TYPE   eventType;
    motBOARD_TYPE   boardType;
    vltBYTES20      eventName;
    } motEVENT_CFG;

/* database for IO signals */
typedef struct
    {
    vltINT32     signal;               /* Signal type                    */
    vltBYTES8    device;               /* Associated device name         */
    vltINT32     startBit;   /* Start bit number                  */
    vltINT32     level;/* Signal logic                     */
    } motSIGNAL_CFG;

typedef struct
    {
    motENC_TYPE   type;              /* Encoder type                      */
    motBOARD_TYPE board;             /* Interface board                   */
    motENC_CODE   code;              /* Code type [binary/BCD/Gray]       */
    vltUINT32     address;           /* Address for external              */
    vltUINT32     resolution;        /* Valid bits (absolute only)        */
    vltUINT32     count;             /* Encoder Counts / Encoder Turn     */
    vltUINT32     circularRange;     /* Circular Range                    */
    vltUINT32     stepCount;         /* Encoder Counts / Motor   turn (STP) */
    vltUINT32     bitShift;          /* Shift (SSI only)                  */
    } motENCODER;

/* database map for special positions’ configuration */
typedef struct
    {
    vltINT32    set;
    motPOS_TYPE posType;
    vltDOUBLE   position;
    motUNIT     unit;                /* Unit must be  motBOARD_UNIT_POS     */
    } motSPECIAL_POSITIONS_CFG;

/* database map for named position offsets in Enc */
typedef struct
    {
        /* Offsets - Unit must be motBOARD_UNIT_POS or default position unit */
    vltDOUBLE maintOffset;/* offset to maintenance section            */
    vltDOUBLE userOffset;/* offset to user defined section            */
    motUNIT   unit;
        /* reserved for internal use - Unit is motBOARD_UNIT_POS */
    vltINT32  maintOffsetEnc;/* Unit is motBOARD_UNIT_POS                */
    vltINT32  userOffsetEnc;/* Unit is motBOARD_UNIT_POS                */
    } motNAMED_POSITION_OFFSET;

/* database map for named positions */
```

```
typedef struct
    {
    vltBYTES32  name;           /* position name                              */
    motPOS_TYPE posType;        /* position type                              */
    vltDOUBLE   obsPos;         /* nominal observation position value         */
    vltDOUBLE   lowRange;       /* lower range offset                         */
    vltDOUBLE   upRange;        /* upper range offset                         */
    vltINT32    maintOffsetAvail;/* maintenance offset available flag         */
    vltDOUBLE   maintOffset; /* offset to maintenance position             */
    vltINT32    userOffsetAvail;/* user offset available flag                 */
    vltDOUBLE   userOffset; /* offset to user position                     */
    motUNIT     unit;           /* position unit                              */
        /* reserved for internal use - Unit is motBOARD_UNIT_POS */
    vltINT32    available;      /* named position is resolved                 */
    vltINT32    obsPosEnc[3];   /* observation positions [low,nom,up]         */
    vltINT32    mntPosEnc[3];   /* maintenance positions [low,nom,up]         */
    vltINT32    usrPosEnc[3];   /* userDefined positions [low,nom,up]         */
    } motNAMED_POSITION;

/* database map for speed */
typedef struct
    {
    vltBYTES32          name;          /* speed name                          */
    vltDOUBLE           speed;      /* speed value                         */
    motUNIT   unit;          /* speed unit                      */
    vltUINT32 pollInt;/* polling interval in ms           */
    vltUINT32 endPollInt;/* end-phase polling interval in ms  */
    vltUINT32 inPosTime;/* needed in-position time for       */
                                       /* motion complete condition in ms   */
        /* reserved for internal use - Unit is motBOARD_UNIT_SPD */
    vltDOUBLE           speedEnc;      /* Speed in motBOARD_UNIT_SPD        */
    } motNAMED_SPEED;

/* database map for two step offset */
typedef struct
    {
    vltDOUBLE      offset;             /* offset value in user unit         */
    motUNIT        unit;               /* Unit must be motBOARD_UNIT_POS    */
    motSPEED       speed;              /* speed to be used for second step  */
    vltINT32       offsetEnc;          /* offset in motBOARD_UNIT_POS       */
    vltDOUBLE      speedEnc;           /* Speed  in motBOARD_UNIT_SPD       */
    } motTWO_STEP_OFFSET;

/* database map for index pulse speed */
typedef struct
    {
    motSPEED       speed;              /* speed to be used for second step  */
    vltDOUBLE      speedEnc;           /* Speed in motBOARD_UNIT_SPD        */
    } motINDEX_PULSE_SPEED;

/* data structures for motor configuration */
typedef struct
    {
    motMOTOR_DEF            motor;
    motBOARD_CONF           boards[motMAX_BOARD];
    motEVENT_CFG            events[motMAX_EVENT];
    motSIGNAL_CFG           signals[motMAX_SIGNAL];
    motAXIS_TYPE            axis;
```

```
    motENCODER                encoder;
    motSPECIAL_POSITIONS_CFG specialPositions[motMAX_SPECIAL_POSITION];
    motNAMED_POSITION_OFFSET offsets;
    vltINT32                  numNamedPositions;
    motNAMED_POSITION        *namedPositions;
    vltINT32                  numNamedSpeeds;
    motNAMED_SPEED           *namedSpeeds;
    motTWO_STEP_OFFSET        twoStepOffset;
    motINDEX_PULSE_SPEED      indexPulseSpeed;
    vltINT32                  defaultSpeed;
    } motCONFIG;

/*
 * status data structure
 */
typedef struct
    {
    motOPMODE      opMode;         /* operational mode                    */
    motMOTIONMODE     motionMode;     /* motion mode                          */
    motMOTION_STATUS  motionStatus;   /* motion status                        */
    vltINT32      motionStep;     /* motion step                          */
    motINIT_STATUS    initStatus;     /* initialization status                */
    vltINT32          initStep;       /* step of initialization sequence      */
    vltINT32          brakeClamped;   /* brake status                         */
    vltINT32        axisEnabled;    /* axis enabled flag                    */
    vltUINT32      eventMask;      /* bit mask to encode abnormal events  */
    vltINT32          amplStatus;     /* amplifier board status               */
    motCONNECTION     powerStatus;    /* pos./neg. relay open closed          */
    vltINT32          overCurrent;    /* Amplifier Current     (over=1)       */
    vltINT32          overTemperature;/* Amplifier Temperature (over=1)       */
    vltINT32          mconStatus;     /* controller board status              */
    vltINT32        driveFault;     /* Drive Fault (Fault=1)                */
    vltINT32        emergencyStop;  /* Emergency Stop (Stopped=1)           */
    vltINT32        posError;       /* Positioning Error (Error=1)          */
    vltINT32        inPos;          /* in position flag                     */
    vltINT32        onLimit;        /* Motor On Limit                       */
    motLIMITS         amplOnLimit;    /* on limit flag for amplifier          */
    motLIMITS         mconOnLimit;    /* on limit flag for controller         */
    vltINT32          mconOnRef;      /* on RefSwitch  for controller         */
    vltINT32          mconOnInd;      /* on IndexPulse for controller         */
    vltINT32          interlock;      /* interlock line status                */
    vltDOUBLE      speed;          /* actual speed in user units          */
    motUNIT      speedUnit;      /* motor speed user unit               */
    vltDOUBLE      current;        /* actual motor current in user units  */
    motUNIT      currentUnit;    /* motor current user unit             */
    vltINT32      posEnc;         /* actual position in encoder ticks    */
    vltDOUBLE         posUser;        /* actual position in user units        */
    motUNIT      posUnit;        /* position user unit                  */
    vltBYTES32        posName;        /* actual position as name              */
    motSECTION        posSection;     /* specifies the section of the name    */
    vltINT32      posIndex;       /* position index                      */
    vltINT32      nextPosEnc;     /* next position                       */
    vltDOUBLE         nextPosUser;    /* next position in user units          */
    vltBYTES32        nextPosName;    /* next position as name                */
    motSECTION        nextPosSection; /* specifies the section of the name    */
    vltINT32      nextPosIndex;   /* next position index                 */
    vltDOUBLE         followingErr;   /* actual following error in user unit */
    ccsTIMEVAL      startTime;      /* start time of last movement in UTC  */
```

```
    ccsTIMEVAL      endTime;      /* end time of last movement in UTC    */
    } motSTATUS;


/* type for user function types */
typedef vltINT32 motUSER_FCT;

/* type for user init function */
typedef ccsCOMPL_STAT (* motINIT_FUNC) ( void *, ccsERROR *);

/* type for user unit conversion function */
typedef ccsCOMPL_STAT (* motCONV_FUNC) ( motMETRIC *, motMETRIC *,
                                         void *, ccsERROR *);

/*
 ******************************************************************************
 * API Function Prototypes
 ******************************************************************************
 */

ccsCOMPL_STAT motInstall        ( char *dbPath, ccsERROR *pError );
ccsCOMPL_STAT motDeinstall      ( char *dbPath, ccsERROR *pError );
ccsCOMPL_STAT motInitDb         ( char *motor, char *file, ccsERROR *pError);

ccsCOMPL_STAT motAttach         ( dbSYMADDRESS dbPath, motACCESS_MODE mode,
  motHANDLE *handle, ccsERROR *error );
ccsCOMPL_STAT motDetach         ( motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motDisable        ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motEnable         ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motSetOpMode      ( const motHANDLE handle[], motOPMODE *mode,
  ccsERROR *error );
ccsCOMPL_STAT motGetOpMode      ( const motHANDLE handle[], motOPMODE *mode,
  ccsERROR *error );

ccsCOMPL_STAT motNameToEnc      ( motHANDLE handle, vltBYTES32 name,
  motSECTION section, vltINT32 *posEnc,
  ccsERROR *error );
ccsCOMPL_STAT motEncToIndex     ( motHANDLE handle, vltINT32 posEnc,
  motSECTION *section, vltINT32 *index,
  ccsERROR *error );
ccsCOMPL_STAT motEncToName      ( motHANDLE handle, vltINT32 posEnc,
  motSECTION *section, vltBYTES32 name,
  ccsERROR *error );
ccsCOMPL_STAT motNameToVel      ( motHANDLE handle, vltBYTES32 name,
                                  vltDOUBLE *velEnc, ccsERROR *error );
ccsCOMPL_STAT motPosToEnc       ( motHANDLE handle, motMETRIC *position,
                                  vltINT32 *encValue, ccsERROR *error );
ccsCOMPL_STAT motEncToPos       ( motHANDLE handle, vltINT32 encValue,
                                  motMETRIC *position, ccsERROR *error );
ccsCOMPL_STAT motVelToEnc       ( motHANDLE handle, motMETRIC *velocity,
                                  vltDOUBLE *encValue, ccsERROR *error );
ccsCOMPL_STAT motEncToVel       ( motHANDLE handle, vltDOUBLE encValue,
                                  motMETRIC *velocity, ccsERROR *error );
ccsCOMPL_STAT motVelToRef       ( motHANDLE handle, motMETRIC *velocity,
                                  vltINT32 *refValue, ccsERROR *error );
ccsCOMPL_STAT motRefToVel       ( motHANDLE handle, vltINT32 refValue,
                                  motMETRIC *velocity, ccsERROR *error );
ccsCOMPL_STAT motDigToCurrent   ( motHANDLE handle, vltINT32 currDig,
```

```
                                  motMETRIC *currUser, ccsERROR *error );

ccsCOMPL_STAT motInitSw         ( const motHANDLE handle[],
  vltLOGICAL mandatory, ccsERROR *error );
ccsCOMPL_STAT motInitHw         ( const motHANDLE handle[],
  vltLOGICAL mandatory, ccsERROR *error );
ccsCOMPL_STAT motSetDefaultUnits( const motHANDLE handle[],
  motDEFAULT_UNITS units[], ccsERROR *error );
ccsCOMPL_STAT motSetMotionMode  ( const motHANDLE handle[],
  motMOTIONMODE mode[], ccsERROR *error );
ccsCOMPL_STAT motMove           ( const motHANDLE handle[], motMOTION *motion,
                                  ccsERROR  *error );
ccsCOMPL_STAT motCheckMove      ( const motHANDLE handle[], motMOTION *motion,
                                  ccsERROR *error );
ccsCOMPL_STAT motSetSection     ( const motHANDLE handle[], motSECTION *section,
  ccsERROR *error );
ccsCOMPL_STAT motGetStatus      ( const motHANDLE handle[], motSTATUS *status,
                                  ccsERROR *error );
ccsCOMPL_STAT motGetDbStatus    ( const motHANDLE handle[], motSTATUS *status,
                                  ccsERROR *error );
ccsCOMPL_STAT motGetDbConfig    ( const motHANDLE handle[], motCONFIG *config,
                                  ccsERROR *error );
ccsCOMPL_STAT motFreeConfig     ( motCONFIG *config );
ccsCOMPL_STAT motWaitInit       ( const motHANDLE handle[], vltINT32 timeout,
                                  motSTATUS *motStatus, ccsERROR *error );
ccsCOMPL_STAT motWaitMove       ( const motHANDLE handle[], vltINT32 timeout,
                                  motSTATUS *motStatus, ccsERROR *error );
ccsCOMPL_STAT motResetSw        ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motResetHw        ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motStop           ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motConnect        ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motDisconnect     ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motClampBrake     ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motUnclampBrake   ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motCheckInterlock ( const motHANDLE handle[], vltINT32 *status,
                                  ccsERROR *error );
ccsCOMPL_STAT motCheckBrake     ( const motHANDLE handle[], vltINT32 *status,
                                  ccsERROR *error );
ccsCOMPL_STAT motInitSampling   ( const motHANDLE handle[], vltINT32 seconds,
                                  motSECTION section, ccsERROR *error );
ccsCOMPL_STAT motStopSampling   ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motLoadPar        ( motHANDLE handle, motBOARD_TYPE board,
                                  vltUINT32 *number, motPARAMETER *parBuf,
                                  ccsERROR *error );
ccsCOMPL_STAT motRetrievePar    ( motHANDLE handle, motBOARD_TYPE board,
                                  vltUINT32 *number, motPARAMETER *parBuf,
                                  ccsERROR *error );
ccsCOMPL_STAT motSetSwLimits    ( const motHANDLE handle[],
  motPOSITION *positions,
  motPOS_TYPE *posTypes,
  motLIMITS *limitFlags,
  ccsERROR *error );

ccsCOMPL_STAT motSetParListAddress ( motHANDLE handle, motUSER_FCT fctType,
                                  vltINT32 fctOccurence, vltBYTES32 fctName,
                                  vltUINT32  parList, ccsERROR *error );
ccsCOMPL_STAT motGetParListAddress ( motHANDLE handle, motUSER_FCT fctType,
                                  vltINT32 fctOccurence, vltBYTES32 fctName,
```

```
                                    vltUINT32 *parList, ccsERROR *error );

ccsCOMPL_STAT motGetHandle      ( char *name, motHANDLE *handle,
  ccsERROR *error );
ccsCOMPL_STAT motGetSemIEV      ( const motHANDLE handle, SEM_ID *semIEV,
  ccsERROR *error );
ccsCOMPL_STAT motWriteEncoderValue ( const motHANDLE handle,
  vltINT32 encoderValue, ccsERROR *error );

/*
 * Tools
 */
void          motVersion          ( vltBYTES80 version );
ccsCOMPL_STAT motClear            ( dbSYMADDRESS dbPath );
ccsCOMPL_STAT motPrintMotors      ( void );
ccsCOMPL_STAT motMotors           ( int *num, vltBYTES20 list[] );
void          motDebug            ( vltINT32 level );

/*
 * Oldies for compatibility
 */
ccsCOMPL_STAT motResetAxis      ( const motHANDLE handle[], ccsERROR *error );
ccsCOMPL_STAT motResetBoards    ( const motHANDLE handle[], ccsERROR *error );

#ifdef __cplusplus
}
#endif

#endif     /* if !MOT_H */

/*************************************************************************/

/*___oOo___*/
```

3.4.2 motDefines.h

```
/*******************************************************************************
* E.S.O. - VLT project
#
# "@(#) $Id: motDefines.h,v 2.7 1998/09/18 14:04:36 vltsccm Exp $"
*
* who       when      what
* --------  --------  ----------------------------------------------
* P.Duhoux  07/07/95  Created  for Release 1.3
* P.Duhoux  25/07/95  Added new literals for optimized unit conversions
*                     motBOARD_UNIT_... motUNIT_... [POS/SPD/CUR/REF]
* P.Duhoux  04/12/95  Added new literal for motion to HOME position
* P.Duhoux  26/03/96  Added macros for position index with section (SPR960183)
* P.Duhoux  26/03/96  Added macros motHALF_TURN_ABS/REL (SPR960512)
* P.Duhoux  13/12/96  Added macros motACTION_DELAY
*/

#ifndef MOT_DEFINES_H
#define MOT_DEFINES_H

/******************************************************************************
 * motDefines.h - MCM definition file
 *
 * This file contains all definitions needed for MCM as #define macros.
 *
 *----------------------------------------------------------------------------
 */
/* formats for dbcfg files */
#define motDBCFG_FMT       "%s.dbcfg"
#define motDBCFG_CWP_FMT   "%s=@%s<alias>%s"
#define motDBCFG_CWP       "<CWP>: "

/* literals for event types */
#define motEVENT_MOTION_END0
#define motEVENT_DRIVE_FAULT       1
#define motEVENT_EMERGENCY_STOP    2
#define motEVENT_OVER_TEMPERATURE       3
#define motEVENT_OVER_CURRENT       4
#define motEVENT_POSITIONING_ERROR 5
#define motEVENT_ON_LIMIT          6
#define motEVENT_NORMAL_STOP            7
#define motEVENT_INTERLOCK_ACTIVE       8
#define motEVENT_NEW_MOTION             9
#define motEVENT_BOARD_INIT_READY      10

/* Public events (mot<event> = 2 << motEVENT_<event>) */
#define motMOTION_END 1
#define motDRIVE_FAULT        2
#define motEMERGENCY_STOP     4
#define motOVER_TEMPERATURE      8
#define motOVER_CURRENT       16
#define motPOSITIONING_ERROR 32
#define motON_LIMIT          64
#define motNORMAL_STOP         128
#define motINTERLOCK_ACTIVE    256
#define motNEW_MOTION          512
#define motBOARD_INIT_READY   1024
```

```
/* logical flags */
#define motNOT_SUPPORTED  0
#define motNOT_APPLICABLE 0
#define motAVAILABLE      1
#define motSET            1

/* empty string */
#define motEMPTY_STRING   ""

/* interlock status strings */
#define motUNLOCKED_STRING "Unlocked"
#define motLOCKED_STRING   "Locked"

/* brake status strings */
#define motUNCLAMPED_STRING "Unclamped"
#define motCLAMPED_STRING   "Clamped"

/* axis status strings */
#define motDISABLED_STRING  "Disabled"
#define motENABLED_STRING   "Enabled"

/* special values */
#define motINVALID -1
#define motNONE     0

/*  8 bit signed extrema */
#define motMIN8  -128
#define motMAX8   127

/* 16 bit signed extrema */
#define motMIN16 -32768
#define motMAX16  32767

/* literal for maximum number of special positions */
#define motMAX_SPECIAL_POSITION  7

/* literal for maximum number of motors to be controlled */
#define motMAX_MOTOR        32

/* literal for maximum number of control boards */
#define motMAX_BOARD         3

/* literal for maximum number of events */
#define motMAX_EVENT            32

/* literal for maximum number of digital IO signals */
#define motMAX_SIGNAL           3

/* literal for minimum number of conversion methods */
#define motMIN_CONV_METHOD      4

/* literal for maximum number of steps in a motion sequence */
#define motMAX_MOTION_SEQ       10

/* literal for minimum polling rate in milli-seconds */
#define motMIN_POLL_RATE       100
#define motMIN_MOVE_POLL_FAR   100
```

```
#define motMIN_MOVE_POLL_NEAR   20

/* literal for maximum number of interpolation points */
#define motMAX_INTERPOL_POINTS  50

/* literal for maximum length of motor name/alias */
#define motMAX_MOTORNAME_LENGTH 12

/* definition of access modes */
#define motUNUSED     0
#define motREAD_ONLY  1
#define motEXCLUSIVE  2
#define motEMERGENCY  3

#define motUNUSED_STRING     "Unused"
#define motREAD_ONLY_STRING  "Read-Only"
#define motEXCLUSIVE_STRING  "Exclusive"
#define motEMERGENCY_STRING  "Emergency"

/* operational modes */
#define motDETACHED       1
#define motNOT_AVAILABLE  2
#define motFIXED_POSITION 3
#define motSIMULATION     4
#define motNORMAL         5
#define motHANDSET        6

#define motDETACHED_STRING       "Detached"
#define motNOT_AVAILABLE_STRING  "NotAvailable"
#define motFIXED_POSITION_STRING "FixedPosition"
#define motSIMULATION_STRING     "Simulation"
#define motNORMAL_STRING         "Normal"
#define motHANDSET_STRING        "Handset"

/* axis types */
#define motAXIS_OFF          motNONE
#define motAXIS_LINEAR       1
#define motAXIS_CIRCULAR     2
#define motAXIS_CIRCULAR_OPT 3

#define motAXIS_OFF_STRING          "Off"
#define motAXIS_LINEAR_STRING       "Linear"
#define motAXIS_CIRCULAR_STRING     "Circular"
#define motAXIS_CIRCULAR_OPT_STRING "CircularOptimized"

/* encoder types */
#define motENC_NONE     0
#define motENC_INC      1
#define motENC_ABS      2
#define motENC_ABS_REL  3

#define motENC_NONE_STRING     "None"
#define motENC_INC_STRING      "Incremental"
#define motENC_ABS_STRING      "Absolute"
#define motENC_ABS_REL_STRING  "Absolute in relative mode"

/* position/speed definition types */
#define motBY_DEFAULT 1
```

```
#define motBY_VALUE   2
#define motBY_NAME    3
#define motBY_INDEX   4

#define motBY_DEFAULT_STRING "ByDefault"
#define motBY_VALUE_STRING   "ByValue"
#define motBY_NAME_STRING    "ByName"
#define motBY_INDEX_STRING   "ByIndex"

#define motNAMED_POSITION_DUMMY "DUMMY"

/* section types */
#define motNO_SECTION   motNONE
#define motOBSERVATION  1
#define motMAINTENANCE  2
#define motUSER_DEFINED 3
#define motALL_SECTIONS 4

#define motNO_SECTION_STRING   "None"
#define motOBSERVATION_STRING  "Observation"
#define motMAINTENANCE_STRING  "Maintenance"
#define motUSER_DEFINED_STRING "UserDefined"
#define motALL_SECTIONS_STRING "AllSections"

/* section with index */
#define motSET_OBSERVATION_INDEX(i) \
    (vltUINT32)((i & 0x00FFFFFF) | (vltUINT32)motOBSERVATION << 24)
#define motSET_MAINTENANCE_INDEX(i) \
    (vltUINT32)((i & 0x00FFFFFF) | (vltUINT32)motMAINTENANCE << 24)
#define motSET_USERDEFINED_INDEX(i) \
    (vltUINT32)((i & 0x00FFFFFF) | (vltUINT32)motUSER_DEFINED << 24)

/* position types */
#define motABSOLUTE          1
#define motRELATIVE          2
#define motLOWER_LIMIT       3
#define motUPPER_LIMIT       4
#define motREFERENCE_SWITCH  5
#define motINDEX_PULSE       6
#define motHOME_POSITION     7
#define motBY_SPEED          8
#define motTWO_STEP_ABS      9
#define motTWO_STEP_REL     10
#define motHALF_TURN_ABS    11
#define motHALF_TURN_REL    12

#define motOFF_POSITION    0x100
#define motOFF_REFERENCE_SWITCH (motOFF_POSITION | motREFERENCE_SWITCH)

#define motABSOLUTE_STRING          "Absolute"
#define motRELATIVE_STRING          "Relative"
#define motLOWER_LIMIT_STRING       "LowerLimit"
#define motUPPER_LIMIT_STRING       "UpperLimit"
#define motREFERENCE_SWITCH_STRING "ReferenceSwitch"
#define motINDEX_PULSE_STRING       "IndexPulse"
#define motHOME_POSITION_STRING     "HomePosition"
#define motBY_SPEED_STRING          "BySpeed"
#define motTWO_STEP_ABS_STRING      "TwoStepAbsolute"
```

```
#define motTWO_STEP_REL_STRING     “TwoStepRelative”
#define motHALF_TURN_ABS_STRING    “HalfTurnAbsolute”
#define motHALF_TURN_REL_STRING    “HalfTurnRelative”

/* motion modes */
#define motINVALID_MOTION_MODE         1
#define motDIRECT_MODE                 2
#define motPOSITION_MODE               3
#define motTRACKING_MODE               4
#define motSPEED_MODE                  5

#define motINVALID_MOTION_MODE_STRING “Invalid”
#define motDIRECT_MODE_STRING         “Direct”
#define motPOSITION_MODE_STRING       “Position”
#define motTRACKING_MODE_STRING       “Tracking”
#define motSPEED_MODE_STRING          “Speed”

/* electrical connection types for amplifier boards */
#define motDISCONNECTED 0
#define motPOSITIVE     1
#define motNEGATIVE     2
#define motBOTH         3

#define motDISCONNECTED_STRING “Disconnected”
#define motPOSITIVE_STRING     “Positive”
#define motNEGATIVE_STRING     “Negative”
#define motBOTH_STRING         “Both connected”

/* board types */
#define motNO_BOARD   0
#define motAMPLIFIER  1
#define motCONTROLLER 2
#define motDIGITAL_IO 3
#define motCPU_BOARD  4
#define motEXT_BOARD  5

#define motNO_BOARD_STRING   “No Board”
#define motAMPLIFIER_STRING  “Amplifier”
#define motCONTROLLER_STRING “Controller”
#define motDIGITAL_IO_STRING “Digital I/O”
#define motCPU_BOARD_STRING  “CPU Board”
#define motEXT_BOARD_STRING  “External Board”

/* limits types */
#define motNOT_ON_LIMIT 0
#define motSW_LOWER     1
#define motSW_UPPER     2
#define motSW_BOTH      3
#define motHW_LOWER     4
#define motHW_UPPER     5
#define motHW_BOTH      6

#define motNOT_ON_LIMIT_STRING “Not on Limit”
#define motSW_LOWER_STRING     “Lower SW Limit”
#define motSW_UPPER_STRING     “Upper SW Limit”
#define motSW_BOTH_STRING      “Both SW Limit”
#define motHW_LOWER_STRING     “Lower HW Limit”
#define motHW_UPPER_STRING     “Upper HW Limit”
```

```
#define motHW_BOTH_STRING      "Both HW Limit"

/* motion status */
#define motSTANDING  1
#define motMOVING    2
#define motTIMEOUT   3
#define motABORTED   4
#define motREDEFINED 5

#define motSTANDING_STRING  "Standing"
#define motMOVING_STRING    "Moving"
#define motTIMEOUT_STRING   "Timeout"
#define motABORTED_STRING   "Aborted"
#define motREDEFINED_STRING "Redefined"

/* initialization status */
#define motNOT_INIT    1
#define motSW_INIT     2
#define motHW_INIT_RUN 3
#define motINIT_DONE   4

#define motNOT_INIT_STRING    "Not Initialized"
#define motSW_INIT_STRING     "S/W Initialized"
#define motHW_INIT_RUN_STRING "H/W Initialization Running"
#define motINIT_DONE_STRING   "Initialization Done"

/* conversion methods types */
#define motEXTERN        1
#define motLINEAR        2
#define motINTERPOLATION 3

/* types for user function */
#define motUSER_INIT 0
#define motUSER_CONV 1

/* unit types */
#define motUNIT_POS    0
#define motUNIT_SPD    1
#define motUNIT_CUR    2
#define motUNIT_REF    3

/* board units */
#define motBOARD_UNIT_POS    "Enc"
#define motBOARD_UNIT_SPD    "Enc/ms"
#define motBOARD_UNIT_CUR    "Dig"
#define motBOARD_UNIT_REF    "Ref"

/* motor types */
#define motDC_MOTOR      1
#define motSTEPPER_MOTOR 2

/* HW initialization actions */
#define motNO_ACTION               motNONE
#define motACTION_END              1
#define motACTION_MOVE             2
#define motACTION_SET_LOW_SW_LIMIT 3
#define motACTION_SET_UP_SW_LIMIT  4
#define motACTION_SET_ENC_VALUE    5
```

```
#define motACTION_CLAMP_BRAKE       6
#define motACTION_DISCONNECT        7
#define motACTION_USERFUNCTION      8
#define motACTION_BOARD_PROCEDURE   9
#define motACTION_DELAY            10

/* encoder coding types */
#define motCODE_BINARY 1
#define motCODE_BCD    2
#define motCODE_GRAY   3

/* events handling types */
#define motNO_EVENT        motNONE
#define motEVENT_INTERRUPT 1
#define motEVENT_SIGNAL    2
#define motEVENT_SOFTWARE  3

/* digital IO signal types */
#define motNO_SIGNAL           -1
#define motSIGNAL_BRAKE_ACTION  0
#define motSIGNAL_BRAKE_STATUS  1
#define motSIGNAL_INTERLOCK     2

#define motSIGNAL_BRAKE_ACTION_NAME  “brakeAction”
#define motSIGNAL_BRAKE_STATUS_NAME  “brakeStatus”
#define motSIGNAL_INTERLOCK_NAME     “interlock”

#define motSIGNAL_ATTRIBUTE_NAME     “value”

#define motSIGNAL_OFF                0
#define motSIGNAL_ACTIVE_LOW         1
#define motSIGNAL_ACTIVE_HIGH        2

/*
 * index of motor status database attributes for cai Access
 */
#define motDB_STATUS_OP_MODE            0
#define motDB_STATUS_MOTION_MODE        1
#define motDB_STATUS_MOTION_STATUS      2
#define motDB_STATUS_MOTION_STEP        3
#define motDB_STATUS_INIT_STATUS        4
#define motDB_STATUS_INIT_STEP          5
#define motDB_STATUS_BRAKE_CLAMPED      6
#define motDB_STATUS_AXIS_ENABLED       7
#define motDB_STATUS_EVENT_MASK         8
#define motDB_STATUS_AMPL_STATUS        9
#define motDB_STATUS_POWER_STATUS      10
#define motDB_STATUS_OVER_CURRENT      11
#define motDB_STATUS_OVER_TEMPERATURE 12
#define motDB_STATUS_MCON_STATUS       13
#define motDB_STATUS_DRIVE_FAULT       14
#define motDB_STATUS_EMERGENCY_STOP    15
#define motDB_STATUS_POS_ERROR         16
#define motDB_STATUS_IN_POS            17
#define motDB_STATUS_ON_LIMIT          18
#define motDB_STATUS_AMPL_ON_LIMIT     19
#define motDB_STATUS_MCON_ON_LIMIT     20
#define motDB_STATUS_INTERLOCK         21
```

```
#define motDB_STATUS_SPEED            22
#define motDB_STATUS_SPEED_UNIT       23
#define motDB_STATUS_CURRENT          24
#define motDB_STATUS_CURRENT_UNIT     25
#define motDB_STATUS_POS_ENC          26
#define motDB_STATUS_POS_USER         27
#define motDB_STATUS_POS_UNIT         28
#define motDB_STATUS_POS_NAME         29
#define motDB_STATUS_POS_SECTION      30
#define motDB_STATUS_POS_INDEX        31
#define motDB_STATUS_NEXT_POS_ENC     32
#define motDB_STATUS_NEXT_POS_USER    33
#define motDB_STATUS_NEXT_POS_NAME    34
#define motDB_STATUS_NEXT_POS_SECTION 35
#define motDB_STATUS_NEXT_POS_INDEX   36
#define motDB_STATUS_FOLLOWING_ERR    37
#define motDB_STATUS_TIME_BEG         38
#define motDB_STATUS_TIME_END         39

/*
 * index of motor configuration database attributes for cai Access
 */
#define motDB_CONF_MOTOR               0
#define motDB_CONF_BOARDS              1
#define motDB_CONF_EVENTS              2
#define motDB_CONF_SIGNALS             3
#define motDB_CONF_AXIS                4
#define motDB_CONF_ENCODER             5
#define motDB_CONF_OFFSETS             6
#define motDB_CONF_SPECIAL_POS         7
#define motDB_CONF_NAMED_POSITIONS     8
#define motDB_CONF_NAMED_SPEEDS        9
#define motDB_CONF_TWO_STEP_OFFSET    10
#define motDB_CONF_INDEX_PULSE_SPEED  11
#define motDB_CONF_DEFAULT_SPEED      12
#define motDB_INIT_ACTIONS            13
#define motDB_UNIT_DEFAULT            14
#define motDB_UNIT_METHODS            15

/***************************************************************************/
/* THE FOLLOWING MACROS REMAIN AVAILABLE BUT ARE NOT USED ANYMORE        */
/*  SINCE THE DIMENSION OF THE TABLES IS FREE                            */
/***************************************************************************/
/* literal for maximum number of named positions */
#define motMAX_NAMED_POSITION   15

/* literal for maximum number of named speeds */
#define motMAX_NAMED_SPEED      15

/* literal for maximum number of HW init actions */
#define motMAX_INIT_ACTION      12

/* literal for maximum number of conversion methods */
#define motMAX_CONV_METHOD      15

#endif /* if !MOT_DEFINES_H */

/***************************************************************************/
```

```
/*___oOo___*/
```

3.4.3 motPublic.h

```
/*******************************************************************************
* E.S.O. - VLT project
*
* "@(#) $Id: motPublic.h,v 2.7 1998/09/18 14:03:29 vltsccm Exp $"
*
* who       when      what
* --------  --------  ----------------------------------------------
* CAM GmbH  --/05/94  Created  for Release 1.1
* P.Duhoux  --/07/94  Modified for Release 1.2
* P.Duhoux  25/01/95  Compliance with OO approach
* P.Duhoux  31/01/95  Added eventIsr in descriptor (Only set by ISR)
* P.Duhoux  15/05/95  Removed field 'motAlias' from the descriptor
*                     - Field 'motName' is now the ALIAS
* P.Duhoux  09/06/95  Modified for CMM Archive
* P.Duhoux  04/07/95  Added literal motMIN_POLL_RATE for status sampling task
* P.Duhoux  07/07/95  Removed all literal definitions (now in motDefines.h
*                     included by mot.h)
* P.Duhoux  25/07/95  Modified motDESCRIPTION for optimized unit conversions
* P.Duhoux  28/08/95  Renamed from motInternal.h
* P.Duhoux  05/12/95  Added flag convConfigDone to indicate the DB config
*                     attributes have been converted to board units
* P.Duhoux  19/01/96  Added defaultUnits and posEnc in descriptor
* P.Duhoux  29/03/96  Added mconSemIEV in motDESCRIPTION
* P.Duhoux  12/11/96  Move definitions to mot.h for motCONFIG data structure
* P.Duhoux  11/12/96  Increased motSEM_TIMEOUT from 5 to 10 seconds (SPR960747)
* P.Duhoux  05/08/97  Removed task option VX_NO_STACK_FILL
*/

#ifndef MOT_PUBLIC_H
#define MOT_PUBLIC_H

/*******************************************************************************
* motPublic.h - motor library public interface file
*
* This header file contains data type definitions and global variables for the
* interface between the mot server and the mot single device libraries. Hence,
* all these sub-modules need to include this file.
*
*-------------------------------------------------------------------------------
*/
#ifdef __cplusplus
extern "C" {
#endif

/*
 * VxWorks include files
 */
#include "vxWorks.h"
#include "semLib.h"
#include "string.h"
#include "ctype.h"
#include "stdio.h"
#include "stdlib.h"
#include "ioLib.h"
#include "symLib.h"
#include "sysSymTbl.h"
```

```
#include "sysLib.h"
#include "taskLib.h"
#include "tickLib.h"
#include "usrLib.h"
#include "errnoLib.h"
#include "vxLib.h"
#include "math.h"

/*
 * MCM include files
 */
#include "mot.h"

#define IN
#define OUT
#define INOUT

/*****************************************************************************
 * server internal macros
 *****************************************************************************/

/* macros for task priorities */
#define motBASE_TASK_PRIOR    (desc->priority)              /* API level      */
#define motINIT_TASK_PRIOR    (motBASE_TASK_PRIOR + 1)      /* H/W Init Task  */
#define motMOVE_TASK_PRIOR    (motBASE_TASK_PRIOR + 2)      /* Motion   Task  */
#define motPOLL_TASK_PRIOR     200                          /* Sampling Task  */

/* macros for task flags */
#define motTASK_FLAGS         (VX_FP_TASK)
#define motINIT_TASK_FLAGS    motTASK_FLAGS                 /* H/W Init Task  */
#define motMOVE_TASK_FLAGS    motTASK_FLAGS                 /* Motion   Task  */
#define motPOLL_TASK_FLAGS    motTASK_FLAGS                 /* Sampling Task  */

/* macros for task stacks */
#define motINIT_TASK_STACK    50000                         /* H/W Init Task  */
#define motMOVE_TASK_STACK    30000                         /* Motion   Task  */
#define motPOLL_TASK_STACK    30000                         /* Sampling Task  */

/* macros for task names */
#define motINIT_TASK_NAME     "tInit_%s"                    /* H/W Init Task  */
#define motMOVE_TASK_NAME     "tMove_%s"                    /* Motion   Task  */
#define motPOLL_TASK_NAME     "tPoll_%s"                    /* Sampling Task  */

/* timeout for <motor descriptor>.accessSem - in seconds */
#define motSEM_TIMEOUT         10
#define motFIS_TIMEOUT          5

#define motsrvSemTimeout  ((int)(motTimeout * motTicksPerSecond))

/* database points */
#define MOTOR_STATUS           "STATUS"
#define MOTOR_SERVER           "SERVER"

/*****************************************************************************
 *                            Internal Data Types
 *****************************************************************************/

/*
```

```
 * index of motor status database attributes for cai Access
 */
typedef enum
    {
    motsrvDB_STAT_OP_MODE = motDB_STATUS_OP_MODE,
    motsrvDB_STAT_MOTION_MODE,
    motsrvDB_STAT_MOTION_STATUS,
    motsrvDB_STAT_MOTION_STEP,
    motsrvDB_STAT_INIT_STATUS,
    motsrvDB_STAT_INIT_STEP,
    motsrvDB_STAT_BRAKE_CLAMPED,
    motsrvDB_STAT_AXIS_ENABLED,
    motsrvDB_STAT_EVENT_MASK,
    motsrvDB_STAT_AMPL_STATUS,
    motsrvDB_STAT_POWER_STATUS,
    motsrvDB_STAT_OVER_CURRENT,
    motsrvDB_STAT_OVER_TEMPERATURE,
    motsrvDB_STAT_MCON_STATUS,
    motsrvDB_STAT_DRIVE_FAULT,
    motsrvDB_STAT_EMERGENCY_STOP,
    motsrvDB_STAT_POS_ERROR,
    motsrvDB_STAT_IN_POS,
    motsrvDB_STAT_ON_LIMIT,
    motsrvDB_STAT_AMPL_ON_LIMIT,
    motsrvDB_STAT_MCON_ON_LIMIT,
    motsrvDB_STAT_MCON_ON_REF,
    motsrvDB_STAT_MCON_ON_IND,
    motsrvDB_STAT_INTERLOCK,
    motsrvDB_STAT_SPEED,
    motsrvDB_STAT_SPEED_UNIT,
    motsrvDB_STAT_CURRENT,
    motsrvDB_STAT_CURRENT_UNIT,
    motsrvDB_STAT_POS_ENC,
    motsrvDB_STAT_POS_USER,
    motsrvDB_STAT_POS_UNIT,
    motsrvDB_STAT_POS_NAME,
    motsrvDB_STAT_POS_SECTION,
    motsrvDB_STAT_POS_INDEX,
    motsrvDB_STAT_NEXT_POS_ENC,
    motsrvDB_STAT_NEXT_POS_USER,
    motsrvDB_STAT_NEXT_POS_NAME,
    motsrvDB_STAT_NEXT_POS_SECTION,
    motsrvDB_STAT_NEXT_POS_INDEX,
    motsrvDB_STAT_FOLLOWING_ERR,
    motsrvDB_STAT_TIME_BEG,
    motsrvDB_STAT_TIME_END,
    motsrvDB_MAX_STAT
    } motsrvDB_STAT;

/* data type for conversion methods */
typedef vltINT32 motCONV_TYPE;

/* data type for HW initialization actions */
typedef vltINT32 motACTION_TYPE;

/* interpolation table fix point */
typedef struct
    {
```

```
    vltDOUBLE x;
    vltDOUBLE y;
    } motPOINT;

typedef struct/* conversion record */
    {
    vltBYTES20   name;
    motCONV_TYPE method;
    vltBYTES32   userFctName;
    vltUINT32    userParList;
    vltDOUBLE    offset;
    vltDOUBLE    slope;
    vltUINT32    numPoints;
    motPOINT     point[motMAX_INTERPOL_POINTS];
    vltUINT32    userFctAddr;
    } motUNIT_CONV;

typedef struct
    {
    motACTION_TYPE actionType;  /* action type (see above)                 */
    vltBYTES32     userFctName;  /* if action USER_FUNCTION: function name  */
    vltUINT32      userParList;   /*   pointer to parameter list             */
    vltINT32       initCode;      /* if action BOARD_PROCEDURE: init code    */
    motMOTIONMODE  motionMode;  /* motion mode                             */
    motPOSITION    position;      /* definition of the position              */
    motPOS_TYPE    posType;  /* position type: ABSOLUTE, RELATIVE, ...  */
    motSPEED       speed;         /* definition of the speed                 */
    } motACTION;

/* database map for special position values (-> SERVER:INTERN) */
typedef struct
    {
    vltINT32       available;
    vltINT32       encValue;            /* Position in motBOARD_UNIT_POS     */
    } motSPECIAL_POSITIONS_INT;

/* data structures for motor descriptor */
typedef struct motDESCRIPTION *motDESCRIPTION_PTR;
typedef struct motDESCRIPTION
    {
    /* access section */
    vltBYTES20         motName;      /* alias to motor point                 */
    int                priority;     /* API task priority                    */
    motACCESS_MODE     accessMode;   /* highest access mode                  */
    SEM_ID             accessSem;    /* semaphore for descriptor access      */
    vltUINT8           numHandles;   /* number of open handles               */
    motOPMODE          opMode;       /* operational mode                     */
    char              *motDbStatus;  /* server database address table STATUS */
    char              *motDbServer;  /* server database address table SERVER */
    struct motDESCRIPTION *next;     /* ptr to next motor descriptor         */

    /* board section */
    vltLOGICAL         mconAvail;    /* flag for "controller available"      */
    vltBYTES8       mconPrefix;   /* prefix of controller SDL             */
    char              *mconFunc;     /* controller SDL routine table         */
    char              *mconDesc;     /* controller descriptor                */
    SEM_ID             mconSemIEV;   /* semaphore for IEV (external encoder) */
```

```
    vltLOGICAL         amplAvail;    /* flag for “amplifier available”       */
    vltBYTES8       amplPrefix;   /* prefix of amplifier SDL              */
    char              *amplFunc;     /* amplifier SDL routine table          */
    char              *amplDesc;     /* amplifier descriptor                 */

    /* axis & encoder section */
    motMOTOR_TYPE      motorType;    /* type of the motor DC/STP             */
    motAXIS_TYPE       axisType;     /* type of the axis LIN/CIR/OPT         */
    motENC_TYPE        encoderType;  /* type of the encoder NO/INC/ABS       */
    int                circularRange;/* range for Circular Axis              */
    vltINT32           chkHwLimit;   /* Check flag for HW limits             */
    vltLOGICAL         logChkHwLimit;/* Log Check flag for HW limits         */

    /* sample section */
    int                sampleTid;    /* task id of sampling task             */
    motSECTION         sampleSection;/* section for sampling / get status    */
    int                pollRate;     /* polling rate in ticks                */
    SEM_ID             pollSem;      /* poll task start semaphore            */
    SEM_ID             pollAck;      /* poll task done  semaphore            */

    /* hwinit/motion section */
    int                motionTid;    /* task id of motion task               */
    ccsERROR          *motionErr;    /* Error stack                          */
    int                hwinitTid;    /* task id of HwInit task               */
    ccsERROR          *hwinitErr;    /* Error stack                          */
    motINIT_STATUS     initStatus;   /* initialization status                */
    vltINT32           initStep;     /* initialization procedure step        */
    motMOTION         *motionSeq;    /* pointer to motion sequence to run    */
    motMOTIONMODE      motionMode;   /* motion mode                          */
    motMOTION_STATUS   motionStatus; /* motion status                        */
    vltINT32           motionStep;   /* number of current motion step        */
    SEM_ID             moveSem;      /* motion task end semaphore            */
    SEM_ID             initSem;      /* semaphore to release motWaitInit     */
    SEM_ID             waitSem;      /* semaphore to release motWaitMove     */
    vltINT32           posEnc;       /* current position                     */
    vltINT32           nextPosEnc;   /* next position                        */
    ccsTIMEVAL       startTime;    /* start time of last motion in ticks   */
    ccsTIMEVAL        endTime;     /* end time of last motoin in ticks     */

    /* digital IO signals access section */
    ioDIRADDRESS       ioSignals[motMAX_SIGNAL];

    /* default unit conversion section */
    vltUINT8           convConfigDone; /* flag to indicate conv config done  */
    vltUINT16          numConvMethods; /* Number of conversion methods       */
    motUNIT_CONV      *convMethods[motMIN_CONV_METHOD]; /* default methods   */
    motDEFAULT_UNITS   defUnits;       /* default units                      */

    /* event handling section            */
    /* KEEP STRICTLY AT END OF STRUCTURE */
    SEM_ID             eventSem;     /* semaphore for abnormal events        */
    vltUINT32          eventMsk;
    vltUINT8           eventFlg[32];
    vltUINT32          eventTab[32];
    } motDESCRIPTION;

/* data type for functions */
typedef ccsCOMPL_STAT (* motFUNCTION) ( void * );
```

```
/* data type for error messages */
typedef vltBYTES256 motERRMSG;

/* data structure for handle description */
typedef struct motHANDLEDATA
    {
    vltUINT32  idFlag; /* own address for identification    */
    motDESCRIPTION       *motDesc;/* pointer to motor descriptor       */
    motACCESS_MODE        accessMode;/* access mode                         */
    struct motHANDLEDATA *next;/* pointer to next handle            */
    } motsrvHANDLEDATA;

/*
 ******************************************************************************
 * Server internal global Variables
 ******************************************************************************
 */
extern SEM_ID            motsrvSemAccess;   /* access protection semaphore   */
extern motsrvHANDLEDATA *motsrvFirstHandle; /* ptr to first motor handle     */
extern motDESCRIPTION   *motsrvFirstDesc;   /* ptr to first motor descriptor */

extern int               motTimeout;        /* semaphore timeout value       */
extern int               motFisTimeout;     /* FIS       timeout value       */
extern int               motTicksPerSecond; /* CPU ticks per second          */

/*
 ******************************************************************************
 * Debug handling
 ******************************************************************************
 */
extern vltINT32              motDebugLevel;

#define motDEBUG_NONE        0x00000000

#define motDEBUG_API         0xF0000000
#define motDEBUG_API_SERVER  0x10000000
#define motDEBUG_API_MOTOR   0x20000000

#define motDEBUG_ACI         0x0F000000

#define motDEBUG_MESSAGE     0x000000F0
#define motDEBUG_MSG_COMMAND 0x00000010
#define motDEBUG_MSG_REPLY   0x00000020
#define motDEBUG_MSG_ERROR   0x00000040

#define motDEBUG_POLLING     0x00000100
#define motDEBUG_MONITOR     0x00000200
#define motDEBUG_CHKSTACK    0x00000400

#define motDEBUG_SDL         0x00F00000
#define motDEBUG_CONTROLLER  0x00100000
#define motDEBUG_AMPLIFIER   0x00200000

#define motDEBUG_DIGITALIO   0x00010000

#define motDEBUG_DRIVER      0x0000000F
#define motDEBUG_WRITE       0x00000001
```

```
#define motDEBUG_READ        0x00000002
#define motDEBUG_COMMAND     0x00000004
#define motDEBUG_INTERRUPT   0x00000008

#define MCM_LOG(fct,txt) \
    { ccsTIMEVAL utc; timsMODE tm; ccsERROR err; vltBYTES32 str; \
      errResetStack(&err); timsGetUTC(&utc,&tm,&err); \
      timsTimeToIsoString(&utc,timsTIME_OF_DAY,6,str,&err); str[15]=0; \
      printf("%s [%s] : %s\n",fct,str,txt); }
#define MCM_LOG1(fct,fmt,p1) \
    { ccsTIMEVAL utc; timsMODE tm; ccsERROR err; vltBYTES256 msg,str; \
      errResetStack(&err); timsGetUTC(&utc,&tm,&err); \
      timsTimeToIsoString(&utc,timsTIME_OF_DAY,6,str,&err); str[15]=0; \
      sprintf(msg,"%s [%s] : ",fct,str); sprintf(str,fmt,p1); \
      printf("%s%s\n",msg,str); }
#define MCM_LOG2(fct,fmt,p1,p2) \
    { ccsTIMEVAL utc; timsMODE tm; ccsERROR err; vltBYTES256 msg,str; \
      errResetStack(&err); timsGetUTC(&utc,&tm,&err); \
      timsTimeToIsoString(&utc,timsTIME_OF_DAY,6,str,&err); str[15]=0; \
      sprintf(msg,"%s [%s] : ",fct,str); sprintf(str,fmt,p1,p2); \
      printf("%s%s\n",msg,str); }
#define MCM_LOG3(fct,fmt,p1,p2,p3) \
    { ccsTIMEVAL utc; timsMODE tm; ccsERROR err; vltBYTES256 msg,str; \
      errResetStack(&err); timsGetUTC(&utc,&tm,&err); \
      timsTimeToIsoString(&utc,timsTIME_OF_DAY,6,str,&err); str[15]=0; \
      sprintf(msg,"%s [%s] : ",fct,str); sprintf(str,fmt,p1,p2,p3); \
      printf("%s%s\n",msg,str); }

#ifdef __cplusplus
}
#endif

#endif /* if !MOT_PUBLIC_H */

/*****************************************************************************/

/*___oOo___*/
```

3.4.4 motciDefines.h

```
/*******************************************************************************
* E.S.O. - VLT project
*
* "@(#) $Id: motciDefines.h,v 1.21 1997/08/08 11:40:13 vltsccm Exp $
*
* who       when      what
* --------  --------  ----------------------------------------------
* P.Duhoux  28/08/95  Created  for Release 1.5
*/

#ifndef MOTCI_DEFINES_H
#define MOTCI_DEFINES_H

/*
 * MACROS for Status Names [Indexes & associated Strings]
 */
#define motciSTAT_OPMODE        0
#define motciSTAT_MOTIONMODE    1
#define motciSTAT_MOTIONSTATUS  2
#define motciSTAT_MOTIONSTEP    3
#define motciSTAT_INITSTATUS    4
#define motciSTAT_INITSTEP      5
#define motciSTAT_BRAKE         6
#define motciSTAT_AXIS          7
#define motciSTAT_EVENTMASK     8
#define motciSTAT_AMPLIFIER     9
#define motciSTAT_POWER        10
#define motciSTAT_CONTROLLER   11
#define motciSTAT_INPOSITION   12
#define motciSTAT_INTERLOCK    13
#define motciSTAT_SPEED        14
#define motciSTAT_CURRENT      15
#define motciSTAT_POSENC       16
#define motciSTAT_POSUSER      17
#define motciSTAT_POSNAME      18
#define motciSTAT_NEXTPOSENC   19
#define motciSTAT_NEXTPOSUSER  20
#define motciSTAT_NEXTPOSNAME  21
#define motciSTAT_FOLLOWINGERR 22
#define motciSTAT_STARTTIME    23
#define motciSTAT_ENDTIME      24
#define motciSTAT_UTC          25

#define motciSTAT_OPMODE_STRING       "OperationalMode"
#define motciSTAT_MOTIONMODE_STRING   "MotionMode"
#define motciSTAT_MOTIONSTATUS_STRING "MotionStatus"
#define motciSTAT_MOTIONSTEP_STRING   "MotionStep"
#define motciSTAT_INITSTATUS_STRING   "InitStatus"
#define motciSTAT_INITSTEP_STRING     "InitStep"
#define motciSTAT_BRAKE_STRING        "Brake"
#define motciSTAT_AXIS_STRING         "Axis"
#define motciSTAT_EVENTMASK_STRING    "EventMask"
#define motciSTAT_AMPLIFIER_STRING    "AmplifierStatus"
#define motciSTAT_POWER_STRING        "PowerStatus"
#define motciSTAT_CONTROLLER_STRING   "ControllerStatus"
#define motciSTAT_INPOSITION_STRING   "InPosition"
```

```
#define motciSTAT_INTERLOCK_STRING    "Interlock"
#define motciSTAT_SPEED_STRING        "Speed"
#define motciSTAT_CURRENT_STRING      "Current"
#define motciSTAT_POSENC_STRING       "PositionEnc"
#define motciSTAT_POSUSER_STRING      "PositionUser"
#define motciSTAT_POSNAME_STRING      "PositionName"
#define motciSTAT_NEXTPOSENC_STRING   "NextPositionEnc"
#define motciSTAT_NEXTPOSUSER_STRING  "NextPositionUser"
#define motciSTAT_NEXTPOSNAME_STRING  "NextPositionName"
#define motciSTAT_FOLLOWINGERR_STRING "FollowingError"
#define motciSTAT_STARTTIME_STRING    "StartTime"
#define motciSTAT_ENDTIME_STRING      "EndTime"
#define motciSTAT_UTC_STRING          "UTC"

#endif
```

4 STRUCTURE AND CONFIGURATION OF THE LOCAL DATABASE

All the routines of the MCM make use of data stored in the LCU local database. The following chapter describes how a motor branch is built up and configured.

4.1 Overview

Each motor to be controlled by MCM must have a dedicated branch in the LCU database. This branch may be located at any place. A motor branch is created by instanciation of a specific database class that describes the physical environment of the motor.

4.2 Classes

A set of standard classes are available, that helps the user to create the database branches needed for the application.

The class definition file **<*path*>/mot/dbl/motMOTORS.class** contains a set of sub-classes of the generic class **motMOTOR**, that corresponds to the various HW configurations supported by MCM.

<u>Code explanation:</u>
From the class name, it is possible to decode the corresponding HW configuration.

mot<*m*><*a*><*v*><*c*><*t*>

m : motor type :	D -> DC-motor	=>	a	in	[X,V]
			v	in	[X,A]
			c	in	[0,M,C]
			t	in	[X,I,A]
	S -> Stepper motor	=>	a	in	[X,V]
			v	in	[X,S]
			c	in	[M,C]
			t	in	[X,S]
a : amplifier :	X -> Stand-alone amplifier	=>	v	set to	X
	V -> ESO-VME4SA/ST	=>	v	in	[A,S]
v : ampl type :	X -> do not care				
	A -> VME4SA-X1				
	S -> VME4ST-STP (Stand-Alone)				
c : controller :	0 -> No controller	=>	t	set to	X
	M -> MAC4	=>	t	in	[I,A,S]
	C -> CAMAC	=>	t	in	[I,A]
t : mcon type :	X -> do not care				
	I -> INC				
	A -> SSI				
	S -> STP				

The following six classes are available:

Class	Motor Type	Amplifier Board	Controller Board	Encoder on Motion Controller
`motDVAMI`	DC	VME4SA-X1	MAC4-INC	Incremental
`motDVAMA`	DC	VME4SA-X1	MAC4-SSI	Absolute

Class	Motor Type	Amplifier Board	Controller Board	Encoder on Motion Controller
`motDXXMI`	DC	Stand-Alone	MAC4-INC	Incremental
`motDXXMA`	DC	Stand-Alone	MAC4-SSI	Absolute
`motSVSMS`	STP	VME4SA-STP	MAC4-STP	Incremental / None
`motDVA0X`	DC	VME4SA-X1	none	None

Their structures differ only from the SDL point (mapping the HW configuration) as shown Figure 3 to Figure 6 below.
These classes are sub-classes of the generic class **`motMOTOR`**; its structure does not contain any instanciation of SDL sub-points as show Figure 7.

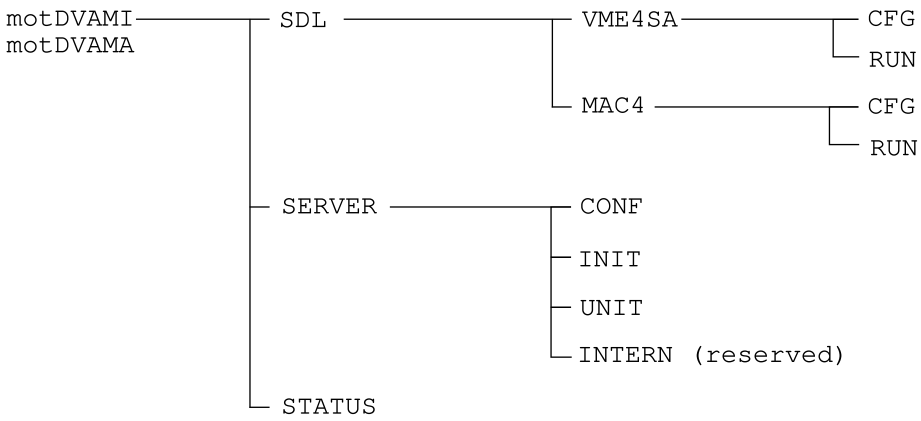


Figure 3 - Classes **`motDVAMI,motDVAMA`**

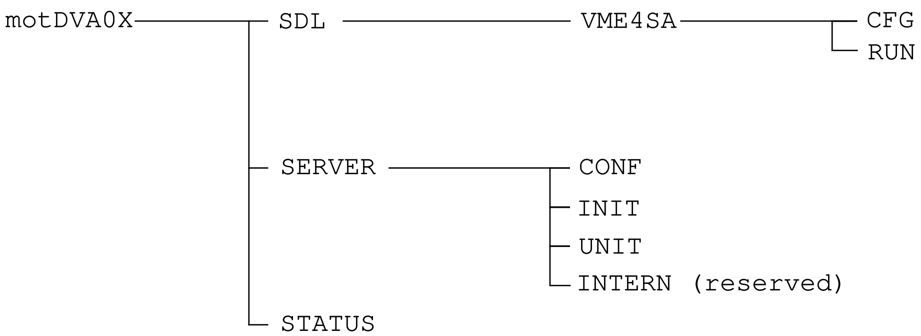


Figure 4 - Class **`motDVA0X`**

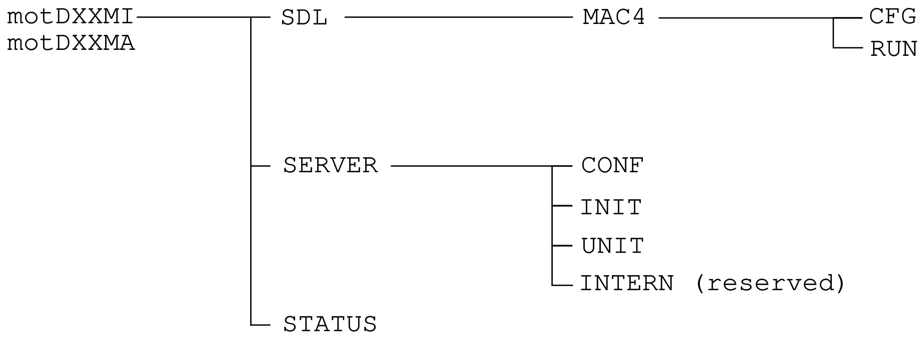


Figure 5 - Classes **motDXXMI,motDXXMA**

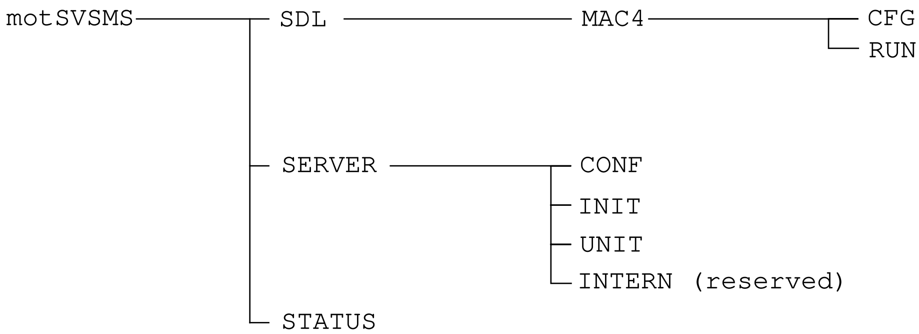


Figure 6 - Class motSVSMS

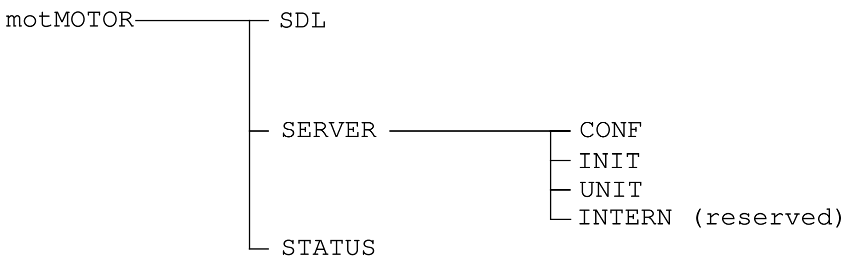


Figure 7 - Generic class **motMOTOR**

4.3 Motor Database Branch Structure

For each motor intended to be controlled by MCM, a database branch must be provided in the local database. The path to this branch is: **<*path*>:<motorName>**. Each motor **must** have a defined **alias** (see [1]), character string of up to 14 characters including the string terminator zero. The definition of aliases for the other sub-points is not mandatory.

The motor database branch is designed so that each SW module refers to its related sub-points only. A motor database branch is divided into three main sections

1. general status information section, which contains all necessary status information about a motor; the DB point of this section is **<*path*>:<motorName>:STATUS**

2. server section, which contains all data needed by the server (Configuration, HW initialization procedure, unit conversion); the DB point of this section is **<*path*>:<motorName>:SERVER**
3. single device library section, which contains all single device specific data; the DB point of this section is **<*path*>:<motorName>:SDL**. This sub-branch is accessed only by the SDL SW modules. This point contains as many sub-points as SDL devices used. Each SDL device sub-point is divided in two parts: **CFG** that contains the configuration parameters, and **RUN** that contains the run-time values (copy of the configuration values + run-time values).

In the next sections, attributes in each section are described more in detail.

Note: All the literals **motXXX** used are defined in **motDefines.h** or **mot.h**.

4.4 Motor Database Branch Configuration

The points **STATUS** (see 4.5), **SERVER:INTERN** (see 4.6.4) and **SDL:<SDL>:RUN** (see 4.7.2) shall not be modified, since their content is reinitialized at installation.

The macros for enumerated values are located in the files **mot.h** or **motDefines.h**.

The following points and attributes must be configured for the motor to operate correctly:

- **SERVER:CONF** This is the main configuration part. (see 4.6.1)

  The following attributes must be preset to appropriate values (These attributes are already preset according to the instanciation class of the motor branch so that they can be defaulted, with exception of the attributes **axis** -preset to **linear**- and **encoder**) :
  - **motor** motor type and timeouts
  - **boards** board configuration description
  - **events** event configuration
  - **signals** signal configuration
  - **axis** axis type
  - **encoder** encoder type and description

  The remaining attributes may not be set (the full usage of the MCM functions is not granted). These attributes are preset so that their content have little effect on MCM:
  - **specialPositions** description of the special positions (limits, etc...)
  - **offsets** position default offsets
  - **namedPositions** named positions description
  - **namedSpeeds** named speeds description
  - **twoStepOffset** offset and speed description for this type of motion
  - **indexPulseSpeed** speed description for fine search of encoder index pulse
  - **defaultSpeed** index of named speed when defaulted
- **SERVER:INIT** This is the description of the HW Initialization Procedure (see 4.6.2)

  No action is preset (**motACTION_END**)
- **SERVER:UNIT** This is the description of all the unit conversion methods (see 4.6.3)

  No method is preset.
  The default units are preset to the board standard units:
  motBOARD_UNIT_POS, **motBOARD_UNIT_SPD** and **motBOARD_UNIT_CUR**.

- **`SDL:<SDL>:CFG`** This is the description of device driver and configuration parameters of the <SDL> board (see 4.7.1, 4.8.1 for the VME4SA board and 4.8.2 for the MAC4 board). The configuration values are the board default values, as set after the board has been reset.

4.5 General Status Section

The structure of this section is a mirror image of the status data structure returned to the user by the **`motGetStatus`** function. For this section, nothing has to be configured. The attributes are of type **`dbSCALAR`** or **`dbVECTOR`** for the **`SCAN`** system to be able to read them.

- **opMode** actual operational mode (enumeration type **`motOPMODE`**):
  - **`motDETACHED`** the motor is not attached (internal)
  - **`motNOT_AVAILABLE`** the motor is not available (status access only)
  - **`motFIXED_POSITION`** the motor is in fixed position (status access only)
  - **`motSIMULATION`** the actions are simulated
  - **`motNORMAL`** the actions are performed normally
  - **`motHANDSET`** the motor is controlled manually (status access only)
- **motionMode** actual motion mode (enumeration type **`motMOTIONMODE`**):
  - **`motPOSITION`** the motor moves in POSITION mode
  - **`motTRACKING`** the motor moves in TRACKING mode
  - **`motSPEED`** the motor moves in SPEED mode
  - **`motDIRECT`** the motor moves in DIRECT mode (amplifier control only)
- **motionStatus** actual motion status (enumeration type **`motMOTION_STATUS`**):
  - **`motSTANDING`** the motor does not move
  - **`motMOVING`** the motor is moving
  - **`motTIMEOUT`** the position could not be reached within the specified time
  - **`motABORTED`** the motor stopped because of an abnormal event
  - **`motREDEFINED`** the motion has been redefined while moving
- **motionStep** step number in a motion sequence
- **initStatus** initialization status (enumeration type **`motINIT_STATUS`**):
  - **`motNOT_INIT`** no initialization done
  - **`motSW_INIT`** motor is SW initialize
  - **`motHW_INIT_RUN`** HW initialization procedure is executing
  - **`motINIT_DONE`** motor is SW & HW initialized; the motor can be moved.
- **initStep** step number of the HW initialization procedure
- **brakeClamped** value of the I/O "brake clamped" signal (see [1])
  possible values are **`ccsTRUE`** or **`ccsFALSE`**
- **axisEnabled** status of the motion control loop:
  possible values are **`ccsTRUE`** when enabled or **`ccsFALSE`** when disabled
- **eventMask** special (abnormal) events; the events are bitwise coded
  - **`motMOTION_END`** the motion is completed
  - **`motDRIVE_FAULT`** a drive fault error has occurred

- **motEMERGENCY_STOP** the emergency stop line is active
- **motOVER_TEMPERATURE** the temperature of the amplifier is too high
- **motOVER_CURRENT** the required current is too high
- **motPOSITIONING_ERROR** the positioning failed
- **motON_LIMIT** the motor has reached a limit (HW or SW)
- **motNORMAL_STOP** the motor shall stop
- **motINTERLOCK_ACTIVE** the interlock line is active
- **motNEW_MOTION** the motion has been redefined while moving
- **motBOARD_INIT_READY** the on-board HW initialization procedure is completed

- **amplStatus** actual amplifier board status **motNOT_AVAILABLE, OK** or **ERROR**
- **powerStatus** state of the electrical connection of the motor (enumeration type **motCONNECTION**):
  - **motDISCONNECTED** the motor is electrically disconnected
  - **motNEGATIVE** only the lower relay of the electrical connection is closed
  - **motPOSITIVE** only the upper relay of the electrical connection is closed
  - **motBOTH** the motor is electrically connected
- **overCurrent** the amplifier current if too high (see also eventMask)
  possible values are **ccsTRUE** or **ccsFALSE**
- **overTemperature** the amplifier temperature is too high (see also eventMask)
  possible values are **ccsTRUE** or **ccsFALSE**
- **mconStatus** actual motion controller board status **motNOT_AVAILABLE, OK** or **ERROR**
- **driveFault** the motion controller is in fault (see also eventMask)
  possible values are **ccsTRUE** or **ccsFALSE**
- **emergencyStop** the emergency stop HW switch has been activated (see also eventMask)
  possible values are **ccsTRUE** or **ccsFALSE**
- **posError** the motion is aborted (see also eventMask)
  possible values are **ccsTRUE** or **ccsFALSE**
- **inPos** the motor has reached the required position
  possible values are **ccsTRUE** or **ccsFALSE**
- **onLimit** the motor is on limit (see also eventMask)
  possible values are **ccsTRUE** or **ccsFALSE**
- **amplOnLimit** amplifier HW limit flag (enumeration type **motLIMITS**):
  - **motNOT_ON_LIMIT**
  - **motHW_UPPER** motor reached upper amplifier HW limit
  - **motHW_LOWER** motor reached lower amplifier HW limit
- **mconOnLimit** motion controller SW/HW limit flag (enumeration type **motLIMITS**):
  - **motNOT_ON_LIMIT**
  - **motSW_UPPER** motor reached upper motion controller SW limit
  - **motSW_LOWER** motor reached lower motion controller SW limit
  - **motHW_UPPER** motor reached upper motion controller HW limit
  - **motHW_LOWER** motor reached lower motion controller HW limit

    - **motHW_BOTH** both HW limits are indicated (for example if cable to motor is disconnected)
- **mconOnRef** motor on Reference switch, possible values are **ccsTRUE** or **ccsFALSE**
- **mconOnInd** motor on encoder index pulse, possible values are **ccsTRUE** or **ccsFALSE**
- **interlock** value of the I/O "interlock" signal (see [1]) possible values are **ccsTRUE** or **ccsFALSE**
- **speed** actual speed of the motor in user speed unit
- **speedUnit** unit of the speed value (is the default speed unit)
- **current** actual motor current in user current unit
- **currentUnit** unit of the motor current value (is the default current unit)
- **posEnc** actual position in encoder ticks
- **posUser** actual position in user position unit (attribute **posUnit**)
- **posUnit** unit of the position values (is the default position unit)
- **posName** actual position name
- **posSection** section where the name was found (enumeration type **motSECTION**):
    - **motOBSERVATION**
    - **motMAINTENANCE**
    - **motUSER_DEFINED**
    - **motNO_SECTION**
- **posIndex** index of the named position in the table (-1 if none)
- **nextPosEnc** next position in encoder ticks
- **nextPosUser** next position in user position unit
- **nextPosName** next position name
- **nextPosSection** section where the name was found (see attribute **posSection**)
- **nextPosIndex** index of the named position in the table (-1 if none)
- **followingErr** actual following error in user unit
- **time** UTC start and end times of the last motion

Note 1: the **ccsTIMEVAL** structure of the UTC times is mapped into a **dbTABLE** structure of one **vltUINT32** for the seconds and one **vltINT32** for the micro-seconds. The first record of the attribute **time** contains the start time, whereas the second the end time of the motion.

4.6 Server Section

This section includes all information about configuration, position and speed parameters, unit conversion and HW initialization of a motor. This section is divided into three subsections.
For each motor to be controlled by MCM, all attributes described in the following have to be set up. Attribute and field names **must not** be changed.

4.6.1 Configuration Section

This section includes the following attributes:

- **motor** describes the motor type and motor dependent parameters; this attribute contains only one record with the following parameters

- type indicates the motor type (enumeration type **`motMOTOR_TYPE`**).
- timeout indicates the time-out value in **`ms`** for a motion step.
- timelim indicates the time-out value in **`ms`** for the motor to move out of limit.
- timemon indicates the period in **`ms`** for automatic status sampling.
- opMode indicates the operational mode on the first attachment.
  - **`motNORMAL`**
  - **`motSIMULATION`**
  - **`motHANDSET`**
  - **`motNOT_AVAILABLE`**
  - **`motFIXED_POSITION`**

__Note:__ The following fields of the attribute **`motor`** are only relevant for the operational mode **`FIXED_POSITION`**; they define the position of the motor to be retrieved by **`motGetStatus`**. This position can not be modified by the API functions.

- posEnc indicates the position value in encoder ticks
- posUser indicates the position value in user units
- posUnit indicates the unit of posUser
- posName indicates the position name
- posSection indicates the corresponding section of posName

The last field is a logical flag that indicates how the HW limits have to be handled:

- chkHwLimit when set tells the SW to check the HW limits for simultaneous activation indicating a possible HW problem (connection broken).

<u>`Instance`</u> DC motor in normal mode, motion timeout is 100s, limit timeout is 5s and the status is refreshed every 1 second, with HW limits check

```
ATTRIBUTE Table SERVER:CONF.motor (1)
BEGIN
    Value ((motDC_MOTOR,100000,5000,1000,motNORMAL,0,0.0,””,””,0,1))
END
```

<u>`Instance`</u> Stepper motor in Fixed-Position mode, set to 50 deg, no HW limit check

```
ATTRIBUTE Table SERVER:CONF.motor (1)
BEGIN
    Value ((motSTEPPER_MOTOR,10000,5000,1000,motFIXED_POSITION,0,50.0,”deg”,””,0,0))
END
```

- **boards** contains all necessary information about the boards, the order __**must be**__ strictly respected, so that index in the table is the board type.
  - type indicates the type of the board (enumeration type **`motBOARD_TYPE`**):
    - **`motAMPLIFIER`**
    - **`motCONTROLLER`**
    - **`motDIGITAL_IO`**
  - available indicates if the board is available **`ccsTRUE`** or **`ccsFALSE.`**

    For the Digital I/O board, the signals will be configured and processed if this field is **`TRUE`**.

- prefix — specifies the prefix of the SDL SW module to be used for interfacing the device driver. If no prefix is given, it will be assumed that the board is not controlled by MCM (Stand-Alone). This applies mainly for the amplifier which must be present (**`available`** must be set to **`ccsTRUE`**), but shall not be accessed by MCM.

  Since no SDL module is associated to the Digital I/O board, the prefix shall be left empty.

<u>`Instance`</u> (for stand-alone amplifier (e.g. Stepper) and no I/O signals)

```
ATTRIBUTE Table SERVER:CONF.boards (motMAX_BOARD)
BEGIN
    Value ((motAMPLIFIER,motAVAILABLE,""),
           (motCONTROLLER,motAVAILABLE,"mac4"),
           (motDIGITAL_IO,motNOT_APPLICABLE,""))
    END
```

<u>`Instance`</u> (for DC motor and I/O signals)

```
ATTRIBUTE Table SERVER:CONF.boards (motMAX_BOARD)
BEGIN
    Value ((motAMPLIFIER,motAVAILABLE,"vme4sa"),
           (motCONTROLLER,motAVAILABLE,"mac4"),
           (motDIGITAL_IO,motAVAILABLE,"/acro0"))
    END
```

- **events** — contains all necessary information about event/signal handling; it is possible to configure up to 32 events; for each event, a literal is defined in the file **`mot.h`**.

For each event to be configured one record has to be defined containing the following fields:

- type — indicates the event type
  - **`motMOTION_END`** — Interrupt generated by the motion controller
  - **`motDRIVE_FAULT`** — Interrupt generated by the motion controller
  - **`motEMERGENCY_STOP`** — Interrupt generated by the motion controller
  - **`motOVER_TEMPERATURE`** — Status bit set by the servo amplifier
  - **`motOVER_CURRENT`** — Status bit set by the servo amplifier
  - **`motPOSITIONING_ERROR`** — Status bit set by the motion controller
  - **`motON_LIMIT`** — Status bit set by the motion controller
  - **`motNORMAL_STOP`** — Software event
  - **`motINTERLOCK_ACTIVE`** — Event set by I/O Signal
  - **`motNEW_MOTION`** — Software event
  - **`motBOARD_INIT_READY`** — Status bit set by the motion controller
- source — indicates the source by which the event is indicated (enumeration type **`motEVENT_TYPE`**):
  - **`motNO_EVENT`** — the event is not handled
  - **`motEVENT_SIGNAL`** — the event is indicated by a signal (see [1] and 2.4.11)
  - **`motEVENT_INTERRUPT`** — the event is flagged by an interrupt (see 2.4.10)
  - **`motEVENT_SOFTWARE`** — the event is detected by software (status polling)

- boardType indicates in case of **motEVENT_INTERRUPT**, by which board the interrupt is generated (enumeration type **motBOARD_TYPE**)
- eventName indicates in case of **motEVENT_INTERRUPT**, the name of the signal as configured by the LCC common software facility

Example:

The event MOTION_END shall be reported by an interrupt generated by the motion controller board:

- type = motMOTION_END
- source = motEVENT_INTERRUPT
- boardType = motCONTROLLER
- eventName = "motionEnd"

Instance

```
ATTRIBUTE Table SERVER:CONF.events (motMAX_EVENT)
BEGIN
    Value (
      (motMOTION_END,    motEVENT_INTERRUPT, motCONTROLLER,"motionEnd"),
      (motDRIVE_FAULT,   motEVENT_INTERRUPT,motCONTROLLER,"driveFault"),
      (motEMERGENCY_STOP,motEVENT_INTERRUPT,motCONTROLLER,"emergencyStop"))
      (motINTERLOCK_ACTIVE,motEVENT_SOFTWARE,motCPU_BOARD,""))
END
```

- **signals** This attribute contains information relative to the three I/O signals:
  - **brakeSignal** acts on the brakes of the motor (clamp/unclamp)
  - **brakeStatusSignal** reports the status of the brake
  - **interlockSignal** reports the state of the interlock line.
- signal indicates the type of the signal (**motSIGNAL_TYPE**)
  - **motNO_SIGNAL** no signal is connected
  - **motSIGNAL_BRAKE_ACTION** signal to clamp/unclamp the brake
  - **motSIGNAL_BRAKE_STATUS** brake status signal
  - **motSIGNAL_INTERLOCK** interlock line signal
- device name of the device which shall process the signals. The standard device is the digital I/O board, but some special applications may use a different board such as the **CAMAC** Motion Controller which can also handle I/O signals.

  The board must be declared as available (see attribute **boards**).

Example: **/acro0** for the **ACROMAG** digital I/O board device (1 device , 64 I/O lines / board).

The two following fields apply only for the standard digital I/O board.

- startBit indicates which I/O line of the device shall be linked to the signal

  The signals are mapped on one line only
- level indicates the logic level of the signal
  - **motSIGNAL_OFF** the signal is disconnected
  - **motSIGNAL_ACTIVE_LOW** the signal is active low

- **motSIGNAL_ACTIVE_HIGH** the signal is active high

Instance

```
ATTRIBUTE Table SERVER:CONF.signals (motMAX_SIGNAL)
BEGIN
    Value (
      (motSIGNAL_BRAKE_ACTION,"/acro0",0,motSIGNAL_ACTIVE_HIGH),
      (motSIGNAL_BRAKE_STATUS,"/acro0",1,motSIGNAL_ACTIVE_HIGH),
      (motSIGNAL_INTERLOCK,   "/acro0",2,motSIGNAL_ACTIVE_HIGH))
END
```

- **axis** indicates the type of the axis (enumeration type **motAXIS_TYPE**):
  - **motAXIS_LINEAR** the axis is linear
  - **motAXIS_CIRCULAR** the axis is circular (no SW nor HW limits)
  - **motAXIS_CIRCULAR_OPT** ditto but optimized

  Instance

```
ATTRIBUTE SERVER:CONF.axis motAXIS_CIRCULAR_OPT
```

- **encoder** Encoder Configuration
  - type Encoder type (enumeration type **motENC_TYPE**)
    - **mot**ENC_NONE no encoder is used (STP only, internal step counting)
    - **mot**ENC_INC the encoder is incremental (INC/STP internal, or external)
    - **mot**ENC_ABS the encoder is absolute (SSI internal, or external encoder)
    - **mot**ENC_ABS_REL the encoder is absolute, but the information is processed as relative (SSI internal, or external encoder)
  - board Interface board (see enumeration type **motBOARD_TYPE**).
    - **mot**CONTROLLER the encoder is connected to the motion controller board
    - **mot**EXT_BOARD the encoder is connected to an external interface board
    - **mot**CPU_BOARD the encoder is connected to the CPU
  - coding Coding type (enumeration type **motENC_CODING**) is not yet used
  - address VME Address where to read the position information, for encoder connected to an external interface board only (must be **ZERO** for **motCPU_BOARD**).
  - resolution Encoder resolution (bits) for Absolute encoder only
  - count Encoder counts / Encoder revolution
  - circularRange Encoder counts / Device turn
  - stepCount Encoder counts / Motor turn (STP only)
  - bitShift Bit shift (Absolute encoder only)

Instance (incremental encoder, stepper motor with 25000 steps/turn, linear axis)

```
ATTRIBUTE Table SERVER:CONF.encoder (1)
BEGIN
    Value ((motENC_INC,motCONTROLLER,motCODE_BINARY,0,15,32767,0,25000,0))
END
```

Instance (absolute encoder, 16 bits, 2 bits right shift, linear axis)

```
ATTRIBUTE Table SERVER:CONF.encoder (1)
BEGIN
    Value ((motENC_ABS,motCONTROLLER,motCODE_BINARY,0,16,65536,0,0,2))
END
```

Instance (stepper 2000 steps/turn, no encoder, circular axis 72000 steps/turn)

```
ATTRIBUTE Table SERVER:CONF.encoder (1)
BEGIN
    Value ((motENC_NONE,0,0,0,0,0,72000,2000,0))
END
```

Instance (incremental encoder supported by the CPU, 16 bits, linear axis)

```
ATTRIBUTE Table SERVER:CONF.encoder (1)
BEGIN
    Value ((motENC_INC,motCPU_BOARD,motCODE_BINARY,0,16,65535,0,0,0))
END
```

Instance (absolute encoder, external encoder interface,24 bits, address **0xAAAE04**)

```
ATTRIBUTE Table SERVER:CONF.encoder (1)
BEGIN
    Value ((motENC_ABS,motEXT_BOARD,motCODE_BINARY,0xAAAE04,24,16777215,0,0))
END
```

- **specialPositions** contains information for the configuration of the special positions (resp. lower and upper SW limits, lower and upper HW limits, Reference Switch, Home Position and Index-Pulse). If the position is not reached during the HW initialization procedure, the positions flagged TRUE are activated as if they had been reached; the associated position is then the one stored below. The position type shall reference an absolute or a relative position (relative to the current position or to one of the **valid** hardware positions). For the hardware positions, the position type shall not reference a SW limit; only **valid** hardware positions shall be referenced (the setting follows the order of the array).
  - setLimit Position set flag (**ccsTRUE** or **ccsFALSE**), if TRUE, the position will be set as declared here.
  - posType Position type (see **motPOS_TYPE** enumeration)
  - posUser Position in user unit
  - unit Position unit

Instance

```
ATTRIBUTE Table SERVER:CONF.specialPositions (motMAX_SPECIAL_POSITION)
BEGIN
    Value (
      (motSET,motLOWER_LIMIT,1.0,"deg"),        set lower SW limit to 1 deg above the
                                                 lower HW limit
      (motNONE,0,0.0,""),                       do not set the upper SW limit
      (motNONE,0,0.0,""),                       do not set the lower HW limit
      (motSET,motABSOLUTE,300,"deg"),           set the upper HW limit to 300 deg
      (motNONE,0,0.0,""),                       do not set the Reference Switch
                                                 position
      (motSET,motREFERENCE_SWITCH,-10.0,"Enc"), set the Home position 10 Enc below REF
```

```
    (motSET,motRELATIVE,100.0,"Enc"))        set the Index Pulse position 100 Enc
                                             above the actual position
END
```

- **offsets** contains the default offsets from observation positions to maintenance and user defined positions, these offsets are used is the offsets are marked not available in the named positions table; one record has to be defined containing the following parameters:
  - maintOffsetindicates the default offset to maintenance positions
  - userOffset indicates the default offset to user defined positions
  - unit indicates the unit of the offset values defined above

The following 2 fields are reserved for internal use only

Example

The default offsets from observation to the maintenance/user defined positions should be resp. 90deg/180deg.

- `maint = 90`
- `user = 180`
- `unit = "deg"`

Instance

```
ATTRIBUTE Table SERVER:CONF.offsets (1)
BEGIN
    Value ((90.0,180.0,"deg",0,0))
END
```

- **namedPositions** contains the definition of the named positions, the table is preset with one empty row; for each named position one record has to be specified. The total number of named positions must be set in the instance.
  - name indicates the position name (up to 31 characters)
  - posType indicates the position type of the named position (enumeration type **`motPOS_TYPE`**)
  - obsPos defines the position value for the observation section
  - lowRange defines the lower range of the named position **relative** to obsPos lower limit of the named position = obsPos - lowRange
  - upRange defines the upper range of the named position **relative** to obsPos upper limit of the named position = obsPos + upRange
  - maintOffsetAvail indicates whether the default offset for the maintenance position should be used or not; possible values are
  - maintOffset contains the offset for the maintenance position, if default one should not be used
  - userOffsetAvail indicates whether the default offset for the user defined position should be used or not
  - userOffset contains the offset for the user defined position, if default one should not be used

- posUnit Position unit for all fields defined above

The following 10 fields are reserved for internal use only

Note: It is recommended to use the filler macro **`motDB_CONF_NPOS_FILLER`** defined in **`motCOMMON.class`** when instanciating the named positions.

Example

The position "`RED`" should be defined; the exact position, lower and upper range should be 100deg/800deg/110deg. For the maintenance position, the default offset should be used, for the user defined position the offset +30deg should be used.

- `name = "red"`
- `posType = motABSOLUTE`
- `obsPos = 100.0`
- `lowRange = 20.0`
- `upRange = 10.0`
- `maintFlag = FALSE`
- `maintOffset = 0`
- `userFlag = TRUE`
- `userOffset = 30.0`
- `posUnit = "deg"`

Instance

```
ATTRIBUTE Table SERVER:CONF.namedPositions (2)
BEGIN
    Value (
      ("red", motABSOLUTE, 100.0,20.0, 10.0,0,  0.0,1,30.0,"deg",motDB_CONF_NPOS_FILLER),
      ("blue",motABSOLUTE,1500.0,50.0,150.0,1,100.0,0, 0.0,"Enc",motDB_CONF_NPOS_FILLER))
END
```

- **namedSpeeds** contains the definition of the named speeds, the table is preset with one empty row; for each named speed one record has to be specified. The total number of named speeds must be set in the instance.
  - name indicates the speed name (up to 31 characters)
  - speed indicates the speed value
  - unit indicates the unit of the speed value
  - pollInt indicates the polling interval (ms) (see 2.4.7)
  - endPollInt indicates the end polling interval (ms)
  - inPosTime indicates the in position time (ms)

The last field is reserved for internal use only.

Example

The speed "`fast`" should be defined; its value should be 10deg/sec. The polling interval during the "far" phase should be 100ms, during the "near" phase 30ms; the motor must be 150ms in position until the status "in position" is set.

- `name = "fast"`
- `value = 10`

- `unit = "deg/sec"`
- `pollInt = 100`
- `endPollInt = 30`
- `inPosTime = 150`

Instance

```
ATTRIBUTE Table SERVER:CONF.namedSpeeds (2)
BEGIN
    Value (
      ("fast",10.0,"deg/sec", 100, 30, 150,0.0),
      ("fine", 2.0,"Enc/ms", 1000,100,2000,0.0))
END
```

- **twoStepOffset** contains the offset and speed definition for **Two Step** type motions.
  - offset Offset
  - unit Offset unit
  - speedHow Speed definition (enumeration type **motSPEC_TYPE**)
  - speedValue Speed value
  - speedUnit Speed unit
  - speedName Speed name
  - speedIndex Speed index
  - pollInt Polling interval (ms)
  - endPollInt End polling interval (ms)
  - inPosTime In position time (ms)

The following 2 fields are reserved for internal use only

Instance

```
ATTRIBUTE Table SERVER:CONF.twoStepOffset (1)
BEGIN
    Value ((10.0,"deg",motBY_NAME,0.0,"","fine",0,0,0,0,0,0.0))
END
```

- **indexPulseSpeed** contains the definition of the speed to be used for the fine search (second step) of the motion to the **Index Pulse**
  - speedHow Speed definition (enumeration type **motSPEC_TYPE**)
  - speedValue Speed value
  - speedUnit Speed unit
  - speedName Speed name
  - speedIndex Speed index
  - pollInt Polling interval (ms)
  - endPollInt End polling interval (ms)
  - inPosTime In position time (ms)

The last field is reserved for internal use only

Instance

```
ATTRIBUTE Table SERVER:CONF.indexPulseSpeed (1)
BEGIN
```

```
    Value((motBY_INDEX,0.0,"","",3,0,0,0,0,0.0))
END
```

- **defaultSpeed** is the index of the named speed to use by default (speedHow = **motBY_DEFAULT**)

  Instance

  ATTRIBUTE SERVER:CONF.defaultSpeed 3

Important Note: the units for the positions (attributes **specialPositions**, **offsets**, **namedPositions** and **twoStepOffset**) **must be** either **Enc** or the default position unit (attribute ***<path>*:SERVER:UNIT.defaultUnits(1,position)**). During installation, SW and HW initialization, the system will attempt to resolve these attributes in board position unit **Enc**; if some special positions or named positions can not be fully resolved by completion of the HW initialization sequence, the motor will not go in the **motINIT_DONE** initialization state, thus no motion can be initiated.

4.6.2 Initialization Section

This section contains the information describing the HW initialization procedure.
The attribute **actions** contains one record for each action to be performed.

- **actions** table of actions (the table is preset with one action, being **motACTION_END**). The total number of actions must be set in the instance.
  - actionType specifies the action type (enumeration type **motACTION_TYPE**)
    - **motACTION_END** the last action of the HW initialization procedure (optional)
    - **motACTION_MOVE** initiate a motion
    - **motACTION_SET_UP_SW_LIMIT** set the upper SW limit
    - **motACTION_SET_LOW_SW_LIMIT** set the lower SW limit
    - **motACTION_SET_ENC_VALUE** set the current position
    - **motACTION_CLAMP_BRAKE** clamp the brake
    - **motACTION_DISCONNECT** disconnect the power lines
    - **motACTION_USERFUNCTION** execute a user-defined function
    - **motACTION_BOARD_PROCEDURE** start an on-board HW initialization procedure
    - **motACTION_DELAY** delay next init step

    The following two fields are required for the action **motACTION_USERFUNCTION**:
  - userFctName User initialization function name (up to 31 characters) (see 2.4.12)
  - userParList Address of user parameter list (see 2.4.12)

    The following field is shared for the actions **motACTION_BOARD_PROCEDURE** and **motACTION_DELAY**:
  - initCode Initialization procedure code / delay in milliseconds

    The following field is required for the action **motACTION_MOVE**:
  - motionMode Motion mode (see enumeration type **motMOTIONMODE**)

    The following fields are required for the actions

    motACTION_MOVE,

    motACTION_SET_UP_SW_LIMIT,

`motACTION_SET_LOW_SW_LIMIT` and
`motACTION_SET_ENC_VALUE` :

- posHow Position definition (see enumeration type **`motSPEC_TYPE`**)
- posValue Position in user unit
- posUnit Position unit
- posName Position name
- posSection Position section
- posIndex Position index
- posType Position type (see enumeration type **`motPOS_TYPE`**)

The following fields are required for the action **`motACTION_MOVE`**:

- speedHow Speed definition (see enumeration type **`motSPEC_TYPE`**)
- speedValue Speed in user unit
- speedUnit Speed unit
- speedName Speed name
- speedIndex Speed index
- pollInt Polling interval (ms)
- endPollInt End polling interval (ms)
- inPosTime In position time (ms)

<u>`Example`</u>

The HW initialization sequence should consist in the six following actions

1. move to the lower HW limit at the named speed "coarse"
2. move to the reference switch at the indexed speed 1
3. initialize current position to 0 deg
4. set lower SW limit to +1000 Enc relative to the lower HW limit
5. move to observation position "red" at the speed 2deg/sec
6. clamp the brake

Accordingly, the attribute **actions** contains seven records set up in the following way (the fields not mentioned are not relevant for the action).

1. record #0
   ```
   actionType = motACTION_MOVE
   motionMode = motSPEED_MODE
   posType = motLOWER_LIMIT
   posHow = motBY_VALUE
   posValue = 0.0
   posUnit = "deg"
   speedHow = motBY_NAME
   speedName = "coarse"
   ```
2. record #1
   ```
   actionType = motACTION_MOVE
   motionMode = motSPEED_MODE
   ```

```
posType = motREFERENCE_SWITCH
posHow = motBY_VALUE
posValue = 0.0
posUnit = "deg"
speedHow = motBY_INDEX
speedIndex = 1
```

3. record #2

```
actionType = motACTION_SET_ENC_VALUE
posType = motABSOLUTE
posHow = motBY_VALUE
posValue = 0.0
posUnit = "deg"
```

4. record #3

```
actionType = motACTION_SET_LOWER_SW_LIMIT
posType = motLOWER_LIMIT
posHow = motBY_VALUE
posValue = +1000
posUnit = "Enc"
```

5. record #4

```
actionType = motACTION_MOVE
motionMode = motPOSITION_MODE
posType = motABSOLUTE
posHow = motBY_NAME
posName = "red"
posSection = motOBSERVATION
speedHow = motBY_VALUE
speedValue = 2.0
speedUnit = "deg/sec"
pollInt = 2000
endPollInt = 200
inPosTime = 1500
```

6. record #5

```
actionType = motACTION_CLAMP_BRAKE
```

7. record #6 (optional)

```
actionType = motACTION_END
```

Instance

```
ATTRIBUTE Table SERVER:INIT.actions (7)
BEGIN
    Value(
        (motACTION_MOVE,"",0,0,motSPEED_MODE,motNONE,0.0,"","",0,0,
         motLOWER_LIMIT,motBY_NAME,0.0,"","coarse",0,0,0,0),
        (motACTION_MOVE,"",0,0,motSPEED_MODE,motNONE,0.0,"","",0,0,
```

```
              motREFERENCE_SWITCH,motBY_INDEX,0.0,””,””,1,0,0,0),
             (motACTION_SET_ENC_VALUE,””,0,0,0,motBY_VALUE,0.0,”deg”,””,0,0,
              motABSOLUTE,motNONE,0.0,””,””,0,0,0,0),
             (motACTION_SET_LOWER_SW_LIMIT,””,0,0,0,motBY_VALUE,1000.0,”Enc”,””,0,0,
              motLOWER_LIMIT,motNONE,0.0,””,””,0,0,0,0),
             (motACTION_MOVE,””,0,0,motBY_NAME,0.0,””,”red”,motOBSERVATION,0,
              motPOSITION_MODE,motBY_VALUE,2.0,”deg/sec”,””,0,2000,200,1500),
             (motACTION_CLAMP_BRAKE,””,0,0,0,0,0.0,””,””,0,0,0,0,0.0,””,””,0,0,0,0),
             (motACTION_END,””,0,0,0,motNONE,0.0,””,””,0,0,0,0,0.0,””,””,0,0,0,0))
END
```

4.6.3 Unit Conversion Section

This section contains the definition of the unit conversion methods to be used.
The first attribute of this section contains the definition of the default user units:

- **defaultUnits** defining the default user units; this attribute contains only one record with the following parameters
  - position indicates the default user unit for position values
  - speed indicates the default user unit for speed values
  - current indicates the default user unit for motor current values

Example

The default user units for position, speed and motor current should be **deg**, **deg/sec** and **mA**

- `position = "deg"`
- `speed = "deg/sec"`
- `current = "mA"`

Instance

```
ATTRIBUTE Table SERVER:UNIT.defaultUnits (1)
BEGIN
    Value((“deg”,”deg/sec”,”mA”))
END
```

Furthermore, for each user unit and at least for the default user units the following unit conversions have to be specified

a. user unit position value to an encoder unit (**Enc**)
b. user unit speed value to an encoder unit (**Enc/ms**)
c. user unit speed value to a velocity reference value (**Ref** for amplifier only)
d. user unit value of the motor current to amplifier (**Dig**) unit

Each unit conversion has to be specified by one record in the table attribute **unitConversion**, whereby origin and destination units are defined in the name stored in the first field of the record.
The names have to be built up in the following way:

- a position *<user position unit>***ToEnc** (a)

- a speed `<user speed unit>`**`ToEnc/ms`** (b)
  `<user speed unit>`**`ToRef`** (c)
- a motor current `<user current unit>`**`ToDig`** (d)

Example

According to the example above the following methods must be provided:

- `degToEnc`
- `deg/secToEnc/ms`
- `deg/secToRef`
- `mAToDig`

Note: All the conversion method shall apply on values and return values of type **`vltDOUBLE`**.

Each record of the attribute **unitConversion** contains the following fields:

- **unitConversion** contains the definition of the conversion methods, the table is preset with four empty rows. The total number of conversion methods must be set in the instance.
  - name indicates the unit conversion (for example “degToEnc”)
  - method indicates the conversion method
    - **`motEXTERN`** the conversion is performed by a user provided function (see 2.4.12).
    - **`motLINEAR`** for a linear transformation
      <destination value> = offset + slope * <origin value>
    - **`motINTERPOLATION`** the destination value is calculated by linear interpolation

  The two following fields are required for the method **`motEXTERN`**.
  - userFctName is the name of the user-defined conversion function
  - userParList contains the address of the data structure to be passed to the user-defined function (see 2.4.12).

  The two following fields are required for the method **`motLINEAR`**.
  - offset is the coordinate at x = 0 in board unit
  - slope is the slope of the line (y = slope * x + offset) in board unit/user unit

  The following 101 fields are required for the method **`motINTERPOLATION`**. They must contain the interpolation table (entries **`(x1,y1)`** to **`(x50,y50)`**), which must define the complete domain of validity of the physical measures (the values are not extrapolated), but all the 50 points do not need to be defined.
  - numPoints indicates the number of entries in the interpolation table; up to 50 entries are allowed (macro **`motMAX_INTERPOL_POINTS`**).
  - x1,y1 first point

  . . .
  - x50,y50 50^{th} point

The last field is reserved for internal use only (should be set to zero).

Important Note: The conversion apply only for Absolute values. For relative positions, dedicated treatment **must** be implemented.

Examples

The conversion of a speed from "`m/s`" to an encoder specific value and vice versa should be done by the user function "**`convertSpeed`**".

- Therefore one record has to be set up containing

```
name = "m/sToEnc/ms"
method = motEXTERN
userFctName = "convertSpeed'
```

The relation between a position value in "`deg`" and the encoder specific unit is linear:

`position[Enc] = 10.0 [Enc] + 200 [Enc/deg] * position [deg]`

- Therefore one record has to be set up containing

```
name = "degToEnc"
method = motLINEAR
offset = 10.0
slope = 200.0
```

Note: the conversion will fail if the **`slope`** is smaller than 10^{-10}.

To get the motor current value in "`mA`" out of the digital one a linear interpolation has to be performed. An interpolation table with 20 entries shall be made available

- Therefore one record has to be set up containing

```
name = "mAToDig"
method = motINTERPOLATION
numPoints = 20
x1 = 1.0        y1 = 0.05
x2 = 2.0        y2 = 0.06
  ....
x20 = 20.0      y20 = 3.576
x21 = 0.0       x21 = 0.0
  ....
x50 = 0.0       y50 = 0.0
```

Note: The conversion will fail for **`x`** values smaller than 1.0 or greater than 20.0 or **`y`** values smaller than 0.05 or greater than 3.576

Note : when instanciating the unit conversion methods, it is recommended to use the filler macros provided in **`motCOMMON.class`** as follows:

- **`LINEAR`** :

`("degToEnc",motLINEAR,"",0,10.0,200.0,motDB_UNIT_LINEAR_FILLER)`

- **INTERPOLATION**:

```
                                       numPoints   1st point  2nd point        20th point
(“mAToDig”,motINTERPOLATION,””,0,0.0,0.0,20,1.0,0.05,2.0,0.06,...,20.0,3.576,
0.0,0.0,...,0.0,0.0,motDB_UNIT_INTERP_FILLER)
21th point    50th point
```

- **EXTERN**:

```
(“m/sToEnc/ms”,motEXTERN,”convertSpeed”,0,0.0,0.0,motDB_UNIT_EXTERN_FILLER)
```

4.6.4 Internal Section

This section is reserved for internal use.

4.7 SDL section

The implementation of this part depends upon the hardware configuration. Its structure is identical for all types of SDL devices. The following rules shall apply:

1. The name of the SDL device point **<u>must</u>** be the name of the associated SDL software module <u>in uppercase</u>.
2. The SDL device point **<u>must</u>** be an instanciation of a sub-class of the generic class **`sdlBASE`**, as described in **`sdlCOMMON.class`**. Specialized sub-classes are available in the class definition files **`mconCOMMON.class`** for a motion controller, and **`amplCOMMON.class`** for an amplifier.
3. The **`RUN`** part **<u>must</u>** contain first the copy of the configuration parameters in the same order as in the **`CFG`** part and then the specific run-time parameters.

4.7.1 CFG section

This section contains the configuration parameters to be used for the SDL board. These values will be loaded onto the board during the SW initialization stage.

- **device** defines the SDL device driver to use for communications with the board.
  - driver name of the driver module
  - number channel number
- **configValues** vector containing the values of all the parameters to be set during SW initialization; these attribute is Read-Only.

4.7.2 RUN section

This section contains the run-time parameters to used by the SDL board. These values are updated every time they are read from/written to the board..

- **currentValues** vector containing the current values of all the parameters of the board. The first part of the vector contains the values of the configuration parameters as set. The second part contains pure run-time parameters.
- **status** vector containing the board status word(s).

For Motion Controllers only,

- **interrupts** table containing the information for all interrupts configured on the board.

4.8 SDL Sections of supported boards

The following sub-sections describe the implementation of the SDL points for the two boards supported by ESO for motion control.

4.8.1 Servo Amplifier VME4SA-X1

This section describes the board parameters for the ESO standard DC motor servo amplifier VME4SA-X1. For the Stepper motors, the ESO standard amplifier VME4ST is not controlled by MCM (Stand-Alone).

4.8.1.1 CFG section

- **device** defines the SDL device driver to use for communications with the board.
  - driver preset to **`ampl`**
  - number channel number
- **configValues** vector containing one value.

Mnemonic	Description	Range Min	Range Max	Preset	Unit
`DO`	DAC Offset	-128	127	0	-

4.8.1.2 RUN section

- **currentValues** vector containing 3 values.

Mnemonic	Description	Range Min	Range Max	Preset	Unit
CFG Parameters					
(see CFG configValues)					
RUN Parameters					
`VR`	Velocity Reference	-255	255	0	-[1]
`MC`	Motor Current	(read only)			

- **status** vector containing 2 status words (Read only).
  - **`AS`** Axis Status
  - **`BS`** Board Status

4.8.2 Motion Controller MAC4

This section describes the board parameters for all hardware types (INC, SSI and STP).

4.8.2.1 CFG section

- **device** defines the SDL device driver to use for communications with the board.
  - driver preset to **`mcon`**
  - number channel number preset to zero (0)

1. **`DO`** & **`VR`** must verify: -128 <= **`VR`**+**`DO`** < 128

- **configValues** vector containing 47 values.

Mnemonic	Description
Motion Configuration (Acceleration/Deceleration)	
`SA`, `SD`	Speed/Brake
`PA`, `PD`	Positioning
`CA`, `CD`	Search Index Coarse (not for SSI nor external absolute encoder)
`IA`, `ID`	Search Index (not for SSI nor external absolute encoder)
`HA`, `HD`	Home mode (not for SSI nor external absolute encoder)
`FA`, `FD`	Find Edge
Test Configuration	
`DA`	DAC Offset
Axis Configuration	
`AX`	Axis Type
`CR`	Circular Range CIR
`LL`	Lower SW Limit LIN
`UL`	Upper SW Limit LIN
Encoder Configuration	
`ET`	Encoder Type
`EB`	Valid Bits (SSI only)
`EC`	Encoder Count
`EO`	Axis Offset (set internally, see `IP`)
`ES`	Step Count (STP only)
`SC`	Bit Shift (SSI only)
Control Loop Configuration (INC & SSI only)	
`GA`	Proportional gain
`ZE`	Zero factor
`IG`	Integral Gain
`IS`	IG Shift Count
`PO`	Pole Factor
`LT`	Limit Torque
Signal Logic Configuration	
`ULL`	Upper limit level
`LLL`	Lower limit level
`RSL`	Reference switch
`DFL`	Drive fault level
STP specific parameters (STP only)	
`MSR`	Motor steps / revolution
`SSF`	Start/Stop frequency
`BST`	Boost time
`CRD`	Creep distance
`SWT`	Servo Wait Time
Auxiliary parameters	
`POL`	Polarity
`WD`	Watchdog time
`MF`	Maximum positioning error
`ED`	Emergency deceleration
`TR`	Target radius
`TT`	Target Settle Time (INC & SSI only)
`SF`	Scaling factor
`LV`	Low Velocity
`IEV`	Interrupt Vector for Encoder Value Interrupt

4.8.2.2 RUN section

- **currentValues**vector containing 72 values.

Mnemonic	Description	
CFG Parameters		
(see CFG configValues)		
RUN parameters		
Interrupt vectors		
`IME`	Interrupt Motion End	
`IDF`	Interrupt Drive Fault	
`IES`	Interrupt Emergency Stop	
Motion		
`SV`	Speed/Brake Velocity	
`AP`, `RP`	Position Abs/Rel	
`PV`	Positioning Velocity	
`CV`	Search Index Coarse Velocity	
`IV`	Search Index Velocity	
`HV`	Home mode Velocity	
`FV`	Find Edge Velocity	
`VT`	Velocity tracking	
`AT`, `RT`	Position tracking	
Encoder Setting		
`EA`	External Address	(see [7], MAC4 User Manual)
Auxiliary Parameters		
`DL`	Define Limit	
Actual State (Read-Only but `IP`)		
`IP`	Init Position (Axis calibration)	
`CP`	Current Position	
`AV`	Actual Velocity	
`EP`	Error Position	
`ECP`	Encoder Counter	(Not updated automatically)
`ECR`	Encoder Counter Raw	(Not updated automatically)
`DAC`	DAC Value	(Not updated automatically)
`CPI`	Current Pos Inc	(Not updated automatically)
`VER`	Board Version[1]	

- **status** vector containing 3 status words (Read-Only)

`USR`	User Status (channel)	(see [4] for detailed explanation)
`SYS`	System Status	(see [4] for detailed explanation)
`COM`	Common Status	(see [4] for detailed explanation)

- **interrupts** table containing 4 rows. (see [4] for detailed explanations)

4.8.2.3 Parameter Range and Preset Value

For each board type (INC, SSI and STP), the following tables gives for each CFG and RUN parameter its range and preset value, used in the generic class definitions.

The units are defined as follows:

1. Bits 0-15 : Firmware version Version.Release,
Bits 16-23 : Hardware version,
Bits 24-31 : Board type : 0 INC, 1 SSI, 2 STP, 3 UNI

AU : Acceleration/Deceleration Unit — Enc/SP/SP (INC/SSI) or Hz/SP (STP)
PU : Position Unit — Enc, ie Encoder Increments (INC/SSI) or Steps (STP).
VU : Velocity Unit — Enc/SP (INC/SSI) or Hz (STP)

where SP is the Sampling Period (10ms for INC/SSI, 32ms for STP).

Detailed information is provided in the MAC4 User Manual (see [7], in German).

CFG Parameter	INC Min	INC Max	INC Preset	SSI Min	SSI Max	SSI Preset	STP Min	STP Max	STP Preset	Unit
SA,SD	1	2E6	50	1	2E6	50	1	8000	1000	AU
PA,PD,FA,FD	1	2E6	50	1	2E6	50	1	8000	500	AU
C/I/H A/D	1	2E6	50	N/A	N/A	N/A	1	8000	500	AU
DA	-2047	2047	0	-2047	2047	0	-50000	50000	0 Hz	–
AX	0	3	1	0	3	1	0	3	1	–
CR	1	1E9	2000	1	EC	2^24	1	10E6	2000	PU
LL	-1E9	1E9	-1E9	0	EC-1	0	-10E6	10E6	-10E6	PU
UL	LL	1E9	1E9	LL	EC-1	2^24-1	LL	10E6	10E6	PU
ET[a]	0	4	1	0	4	2	0	4	1	–
EB[b]	8	32	15	8	32	24	8	32	15	–
EC	1	2E9	32768	1	2E9	2^24	1	2E9	32768	PU
EO	-1E9	1E9	0	-1E9	1E9	0	-10E6	10E6	0	PU
ES	N/A	N/A	N/A	N/A	N/A	N/A	1	1E6	1000	PU
SC	N/A	N/A	N/A	0	24	0	N/A	N/A	N/A	–
GA	0	255	7	0	255	7	N/A	N/A	N/A	–
ZE	0	255	232	0	255	232	N/A	N/A	N/A	–
IG	0	255	0	0	255	0	N/A	N/A	N/A	–
IS	0	8	8	0	8	8	N/A	N/A	N/A	–
PO	0	255	0	0	255	0	N/A	N/A	N/A	–
LT	1	2047	2047	1	2047	2047	N/A	N/A	N/A	–
ULL,LLL,RSL,DFL[c]	0	2	1	0	2	1	0	2	1	–
MSR	N/A	N/A	N/A	N/A	N/A	N/A	1	10E6	25000	Step
SSF	N/A	N/A	N/A	N/A	N/A	N/A	1	2000	100	Hz
BST	N/A	N/A	N/A	N/A	N/A	N/A	1	10000	100	ms
CRD	N/A	N/A	N/A	N/A	N/A	N/A	0	30000	100	PU
SWT	N/A	N/A	N/A	N/A	N/A	N/A	0	3.6E6	0	ms
POL[d]	0	1	0	0	1	0	0	1	0	–
WD	0	3.6E6	0	0	3.6E6	0	0	3.6E6	0	ms
MF	1	32000	8000	1	32000	8000	1	32000	8000	PU
ED	1	2E6	50	1	2E6	50	1	8000	5000	AU
TR	0	32000	0	0	32000	0	0	32000	0	PU
TT	0	3.6E6	0	0	3.6E6	0	N/A	N/A	N/A	ms
SF[e]	0	16	0	0	16	0	0	12	0	–
LV	1	2E6	4	1	2E6	4	1	30000	30	VU
IEV	0x00	0x00	0xFF	0x00	0x00	0xFF	0x00	0x00	0xFF	

a. 0 is internal count (STP only), 1 is internal incremental encoder (INC and STP), 2 is internal absolute encoder (SSI only), 3 is external relative encoder and 4 is external absolute encoder.
b. For INC and STP, preset to 15 for internal encoder. To be set for external encoder (see [7]).
c. 0 is Off, 1 is active Low, 2 is active High
d. Output polarity (0 is Normal, 1 is Reverse)
e. The dividing factor 2^{SF} is applied to all Acceleration, Deceleration and Velocity parameters.

RUN Parameter	INC Min	INC Max	INC Preset	SSI Min	SSI Max	SSI Preset	STP Min	STP Max	STP Preset	Unit
SV	-250E6	250E6	1000	-250E6	250E6	1000	-50000	50000	4000	VU
AP,RP	-1E9	1E9	0	-1E9	1E9	0	-10E6	10E6	0	PU
PV	1	250E6	1000	1	250E6	1000	1	50000	4000	VU
CV,IV,HV	-250E6	250E6	1000	N/A	N/A	N/A	-50000	50000	2000	VU
FV	-250E6	250E6	1000	-250E6	250E6	1000	-50000	50000	2000	VU
VT	-250E6	250E6	100	-250E6	250E6	100	-50000	50000	2000	VU
AT,RT	-1E9	1E9	0	-1E9	1E9	0	-10E6	10E6	0	PU
EA[a]	$3000	$BEFFFE		3000	$BEFFFE		$3000	$BEFFFE		Hex
DL[b]	0	2	0	0	2	0	0	2	0	-
IP	-1E9	1E9	No	-1E9	1E9	No	-10E6	10E6	No	PU

a. This parameter can be set to access the encoder information delivered by an external encoder interface board on the motion controller board (ET set to 3 or 4). Its value depends on the device channel (**$330A**, **$370A**, **$3B0A** and **$3F0A** for DPRAM access for the channels 1 to 4 respectively), or any valid VME-bus address, see[7]. For internal encoder, the address is preset at reset to **$F00001**, **$F00011**, **$F00021**, **$F00031** for the channels 1 to 4 respectively, see [7] .
b. 0 is Reference Switch, 1 is Lower HW Switch, 2 is Upper HW Switch)

Independently on the board type, all parameters may be preset in the database. The parameters which shall not be loaded to the board are ignored.

<u>**Note :**</u> Some parameters are duplicata of parameters defined in the **SERVER:CONF** section: they allow the **mac4** module to be operated without the **mot** module (maintenance, test or special application). Concerned are the parameters **AX**, **ET**, **EA**, **EB**, **ES**, **SC** and **CR**.

5 INSTALLATION GUIDE

5.1 Installation Requirements

5.1.1 Hardware Requirements

In addition to the default LCU hardware (see [6]), the following boards are required for MCM:

- Motion controller board (optional):
  MACCON MAC4-INC, Version 4.1A or higher (recommended is 4.2A)
  MACCON MAC4-SSI, Version 4.1A or higher (recommended is 4.2A)
  MACCON MAC4-STP, Version 2.0A or higher
- Digital I/O board (optional):
  ACROMAG, Series 948x
- Time Interface Module TIM (optional)
- Amplifier board:
  ESO VME4SA-X1 4-Channel DC Servo Amplifier, any version
  ESO VME4ST 4-Channel STP, any version
  or any other board not controlled by MCM, but compatible with the motion controller (in this case, the motion controller is mandatory).

5.1.2 Software Requirements

- VLT Common Software, Release **`OCT98`** (from **`DEC95`**, but for the device driver **mcon**).

5.2 Building the Software

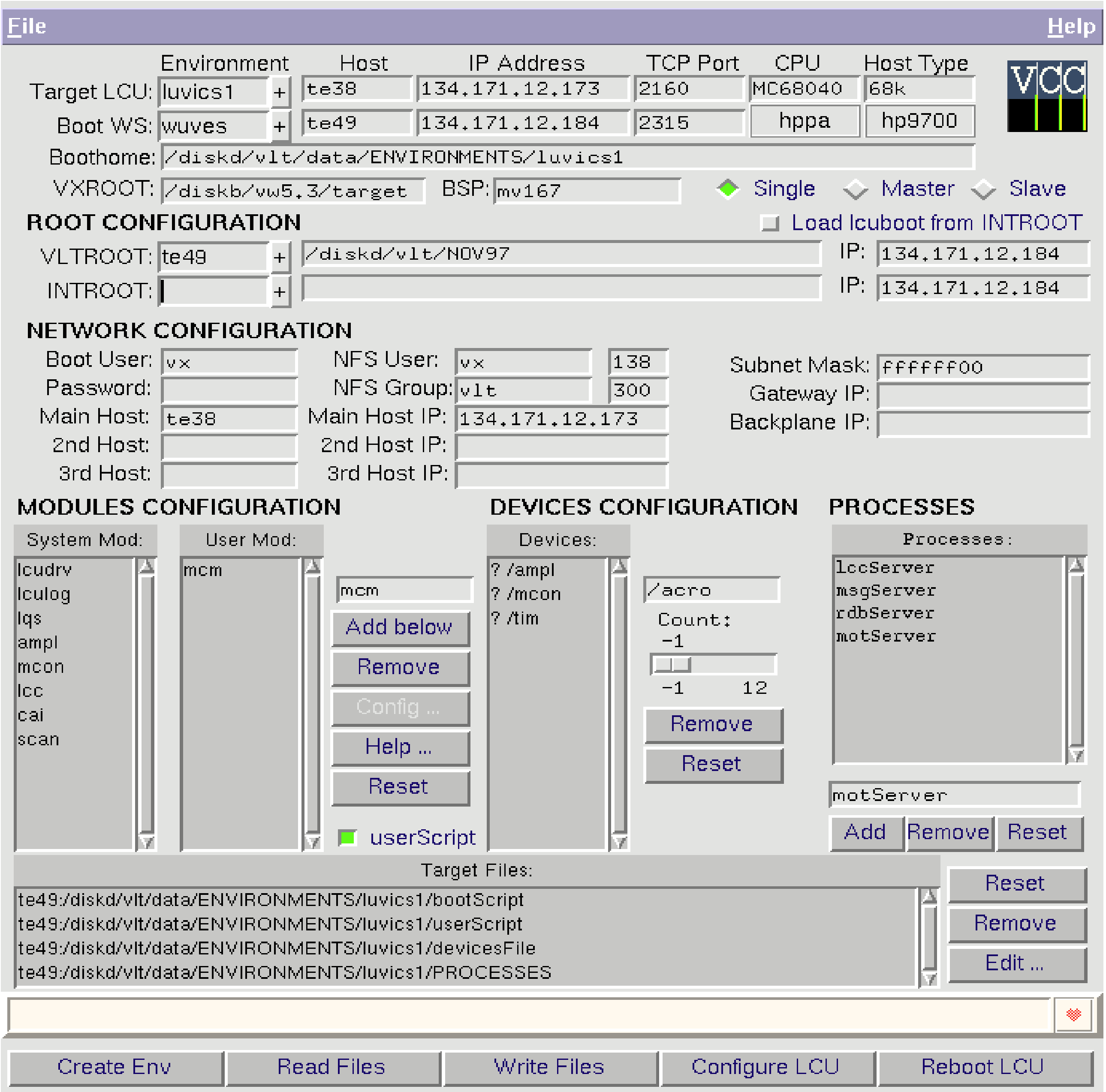
The MCM modules are part of the VLT Common Software, thus their generation is part of the standard installation procedure, as described in [6].

5.3 LCU Environment Configuration

The configuration of the already existing LCU environment (directory **`$VLTDATA/ENVIRONMENTS/$LOCALENV`**) is performed using the tool **`vccConfigLcu`** tool exclusively (see [8]).
In order to get the MCM modules loaded (see **`vccConfigLcu`** panel below):

1. Click the mouse in the field **`User Mod.`**
2. Type in the following name, entering the input by clicking on **`Add Below`**:

   `mcm`
3. Check if the **`motServer`** process is in the list of **`PROCESSES`** .
4. Save the configuration (button **`Write Files`**, do overwrite the already existing files)
5. **`Configure LCU`**.



5.3.1 Local Database

Before the LCU shall be rebooted, the appropriate database must be generated. Therefore, the file **$VLTDATA/ENVIRONMENTS/$LOCALENV/dbl/DATABASE.db** must be edited.
A branch config file (BCF) may be included, the file must have been installed under **$XXXROOT/dbl** or **$XXXROOT/vw/dbl** (where **XXXROOT** is **VLTROOT** or **INTROOT**):

<u>Example:</u>

```
#define motNUMBER_NAMED_POSITIONS 10
#define motNUMBER_NAMED_SPEEDS 5
#define motNUMBER_INITHW_STEPS 3
#define motNUMBER_CONV_METHODS 4
```

```
#define motMOTORROOT  :Instr1:Motors:Motor1
#define motMOTORTYPE  motDVAMI
#define motMOTORNAME  M1

#include "motor.db"
```

The DB branch **:Instr1:Motors:Motor1** will be created that contains the default values as for the class DVAMI. Using the Engineering Tool **motei**, one can modify/customize the configuration of the motor. When the motor has been fully configured and tested, the configuration file **M1.db-cfg** shall be generated and stored in the **./config** path of the application module. This file shall then be installed in the **$INS_ROOT/config** path and shall be read from **motServer** at installation time by **motInitDb(1)** before invoking **motInstall(1)**.

Important Note: When instanciating a motor, one should pay attention to the number of entries for the NamedPositions, NamedSpeeds, InitializationProcedure and UnitConversionMethods. At run-time, it is not possible to add any new entry but solely modify the existing ones. By default, when also no number of entries is specified, the numbers of entries are defaulted to **motMAX_NAMED_POSITION=15**, **motMAX_NAMED_SPEED=15**, **motMAX_INIT_ACTION=12** and **motMIN_CONV_METHOD=4** (see **motDefines.h**).

The MCM branch(es) may also be fully described as shown below; however, this method is not any more recommended since one would have to regenerate the database each time an attribute is modified.

Example:

Create a motor **Motor1** as an instance of the class **motDVAMI**, but circular axis and unit conversions; its absolute path shall be **:Instr1:Motors:Motor1** (**Do not forget to assign a valid alias, here M1**). This description may be put as content of the BCF **motor.db** as for the previous example.

```
#include "motMOTORS.class"

POINT BASE_CLASS :Instr1:Motors
BEGIN

ATTRIBUTE motDVAMI Motor1
BEGIN
    ALIAS "M1"
    //
    // Axis Type
    //
    ATTRIBUTE int32 SERVER:CONF.axis motAXIS_CIRCULAR
    //
    // Unit Conversion Methods
    //
    ATTRIBUTE Table   SERVER:UNIT.unitConversion (4)
    BEGIN
        Value (
        ("degToEnc",motLINEAR,"",0,0.0,200.0,motDB_UNIT_LINEAR_FILLER),
          ("deg/secToEnc/ms",motLINEAR,"",0,0.0,0.2,motDB_UNIT_LINEAR_FILLER),
        ("deg/secToRef",motLINEAR,"",0,0.0,0.2,motDB_UNIT_LINEAR_FILLER),
        ("mAToDig",motLINEAR,"",0,0.0,50.8,motDB_UNIT_LINEAR_FILLER))
    END
END
```

All the attributes may be overloaded, sub-classes may be created, etc....
The attributes of type **TABLE** (**SERVER:CONF.namedPositions**, **SERVER:CONF.namedSpeeds**,

SERVER:INIT.actions and **SERVER:UNIT.unitConversion**) are defaulted with 1 record.

Examples are available under the module directory **mot/lcu/dbl**.

The following command, using the **db** target , generates the LCU database:

```
% make db
```

When the database has been successfully generated, the last step of the LCU configuration may be performed from the panel **vccConfigLcu**:

7. **Reboot LCU**

Set the connection time to about 180 seconds, and check that the procedure completes successfully.

8. Close the connection and quit the panel.

5.3.2 Verification

Functions performed during the installation phase always log 2 (for **SUCCESS**) or 1 (for **FAILURE**) on the console.

The tools, described in section 3.3, can be used to test that the motor library has been installed correctly. From the VxWorks Shell, issue the following commands, which display the module version on the console:

```
lte32->mcmVersion

mcm    - Motor Control Module          Revision: 2.7  - OCT98

motci  - Motor Control Module CI       Version 2.1  - JUL98
mot    - Motor Control Module API      Version 2.7  - SEP98
sdl    - SDL Common Module             Version 1.11 - MAR98
mac4   - Motion Controller SDL         Version 3.2  - JUL98
vme4sa - Servo Amplifier SDL           Version 2.2  - MAR98

value = 1 = 0x1
lte32->
```

The version of each module may also be retrieved:

```
lte32->motVersion

mot    - Motor Control Module API      Version V.VV - MMMYY

value = 2 = 0x2 =  + 0x1
lte32->motciVersion

motci  - Motor Control Module CI       Version V.VV - MMMYY

value = 2 = 0x2 =  + 0x1
lte32->sdlVersion

sdl    - SDL Common Module             Version V.VV - MMMYY[1]

value = 2 = 0x2 =  + 0x1
lte32->mac4Version
```

1. Version identifier: ***V.VV*** is the version number e.g. **1.12**, and ***MMMYY*** the date (month/year) e.g **NOV97**

```
mac4   - Motion Controller SDL          Version V.VV - MMMYY

value = 2 = 0x2 =  + 0x1
lte32->vme4saVersion

vme4sa - Servo Amplifier SDL            Version V.VV - MMMYY

value = 2 = 0x2 =  + 0x1
lte32->
```

In order to verify the installation of the ACI, invoke the CCS Engineering Interface **ccseiMsg**, select the correct environment and the process **`motServer`** and enter the command **`VERSION`**. The reply should contain the two following lines:

```
VERSION mot    - Motor Control Module API       Version V.VV - MMMYY
VERSION motci  - Motor Control Module CI        Version V.VV - MMMYY
```

The version may also be retrieved directly from the Unix shell (from known **`RTAP`** environment):

```
pduhoux@te49:~/MCM 58->msgSend -n lted motServer "VERSION" ""
MESSAGEBUFFER:
mot    - Motor Control Module API       Version V.VV - MMMYY
MESSAGEBUFFER:
motci  - Motor Control Module CI        Version V.VV - MMMYY
pduhoux@te49:~/MCM 59->
```

After installation of motors, one can check the motors which have been installed:

```
luvics1->motPrintMotors

Installed Motors
================================================================================
|      Motor Alias     | Database absolute path
--------------------------------------------------------------------------------
| DVAMI                | :MOTEI:DVAMI
| DPOSM                | :Appl_data:uves:ins:preSlit:depolarizer:slide:MOTOR
| DPORM                | :Appl_data:uves:ins:preSlit:depolarizer:rotate:MOTOR
| SVSMS                | :MOTEI:SVSMS
| DXXMA                | :MOTEI:DXXMA
| DXXMI                | :MOTEI:DXXMI
| DVAMA                | :MOTEI:DVAMA
| DVA0X                | :MOTEI:DVA0X
================================================================================

value = 2 = 0x2
```

6 ERROR MESSAGES AND RECOVERY

The error messages files **`mot_ERRORS`** and **`motci_ERRORS`** are located in the **`$VLTROOT/ERRORS`** directory, and in the respective **`./ERRORS`** directory of the **mot** and **motci** modules.
The literals associated to the error codes are located in the files **`motErrors.h`** and **`motciErrors.h`** in the **`$VLTROOT/include`** directory, and in the respective **`./include`** directory of the **mot** and **motci** modules.

For detailed information about the LCC standard error mechanism see [1].

APPENDIX A

This appendix contains drawings of all required motor control configurations as far as known yet.
Important Note: All the following configurations but #21 to #23 show the Servo-Amplifier as a VME-bus board (namely the ESO VME4SA standard amplifier). As a matter of fact, the amplifier can be replaced by any kind of Maccon MAC4-compatible purely passive amplifier.

Config1: amplifier, controller, two sets of limit switches, reference switch, incremental encoder with zero pulse

Config2: amplifier, controller, two sets of limit switches, reference switch, incremental encoder without zero pulse

Config3: amplifier, controller, two sets of limit switches, reference switch, absolute encoder

Config4: amplifier, controller, two sets of limit switches, no reference switch, incremental encoder with zero pulse

Config5: amplifier, controller, two sets of limit switches, no reference switch, incremental encoder without zero pulse

Config6: amplifier, controller, two sets of limit switches, no reference switch, absolute encoder

Config7: amplifier, controller, one set of limit switches, no reference switch, incremental encoder with zero pulse

Config8: amplifier, controller, one set of limit switches, no reference switch, incremental encoder without zero pulse

Config9: amplifier, controller, one set of limit switches, no reference switch, absolute encoder

Config10: amplifier, controller, one set of limit switches, reference switch, incremental encoder with zero pulse

Config11: amplifier, controller, one set of limit switches, reference switch, incremental encoder without zero pulse

Config12: amplifier, controller, one set of limit switches, reference switch, absolute encoder

Config13: amplifier, controller, no limit switches, no reference switch, incremental encoder with zero pulse

Config14: amplifier, controller, no limit switches, reference switch, incremental encoder without zero pulse

Config15: amplifier, controller, no limit switches, reference switch, absolute encoder

Config16: amplifier, controller, data conversion module, two sets of limit switches, reference switch, arbitrary encoder

Config17: amplifier, controller, data conversion module, two sets of limit switches, no reference switch, arbitrary encoder

Config18: amplifier, controller, data conversion module, one set of limit switches, reference switch, arbitrary encoder

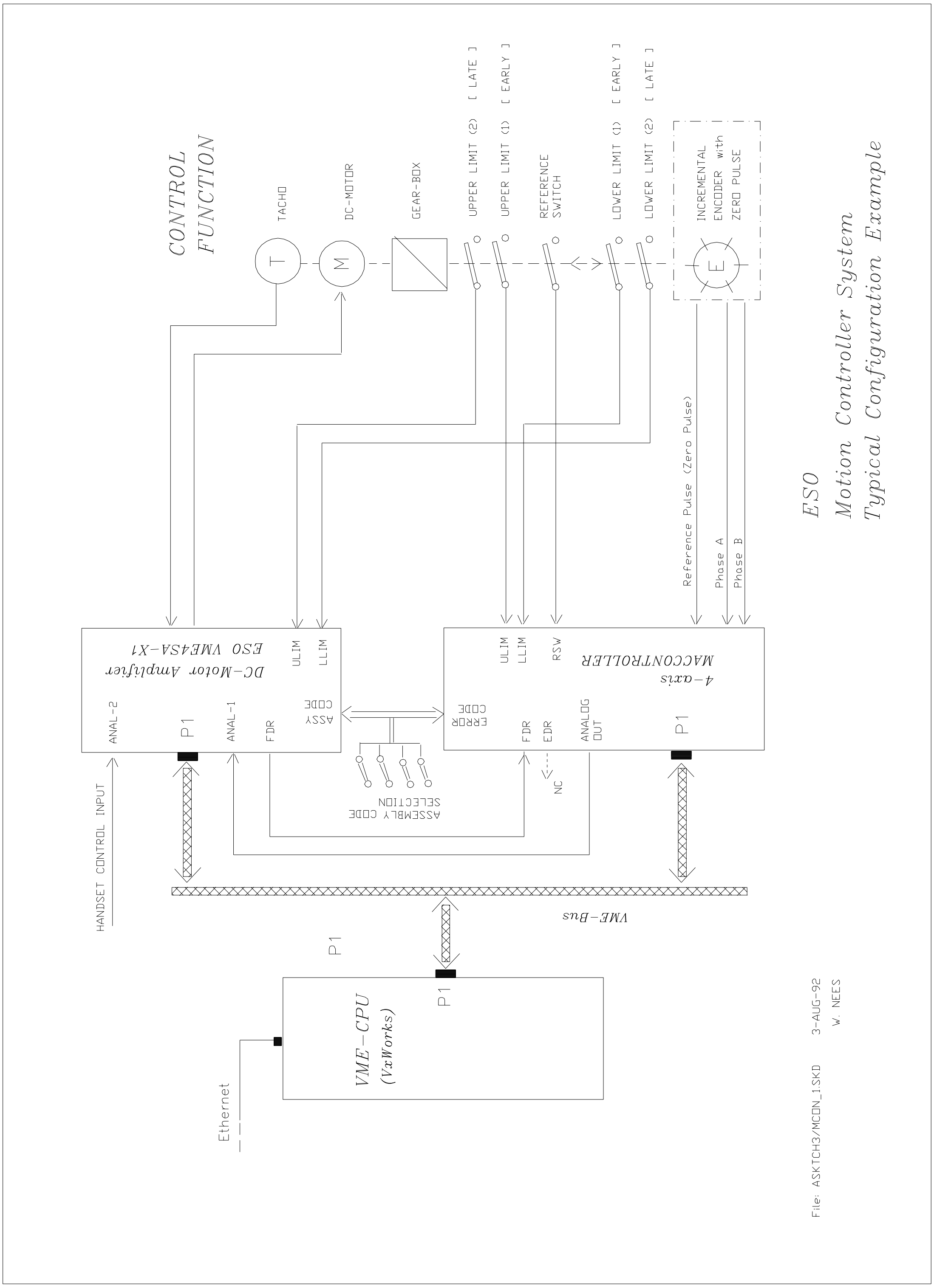
Config19: amplifier, controller, data conversion module, one set of limit switches, no reference switch, arbitrary encoder

Config20: amplifier, controller, data conversion module, no limit switches, no reference switch arbitrary encoder

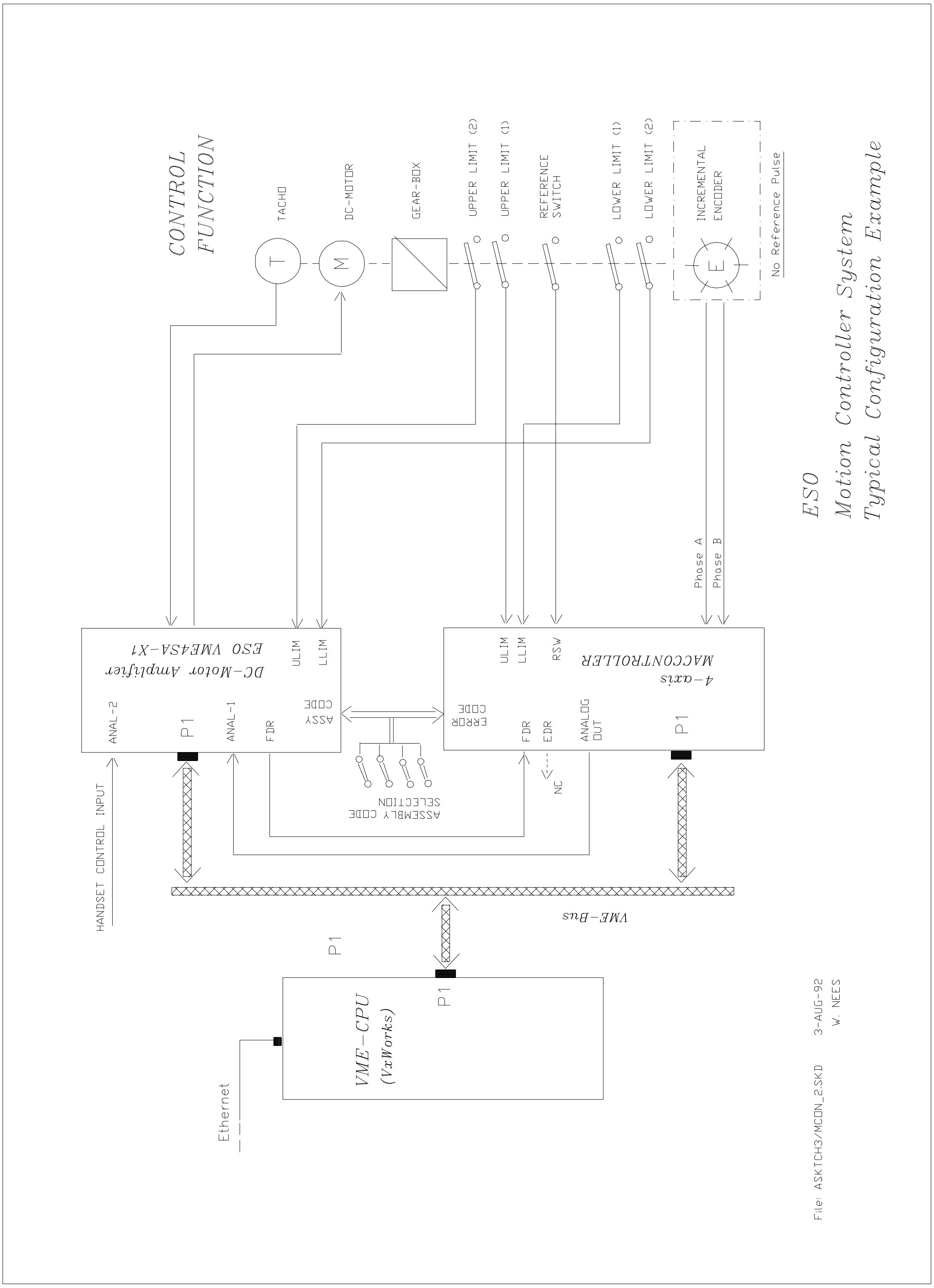
Config21: amplifier, I/O status module, two sets of limit switches, reference switch, no encoder

Config22: amplifier, one set of limit switches, no reference switch, no encoder

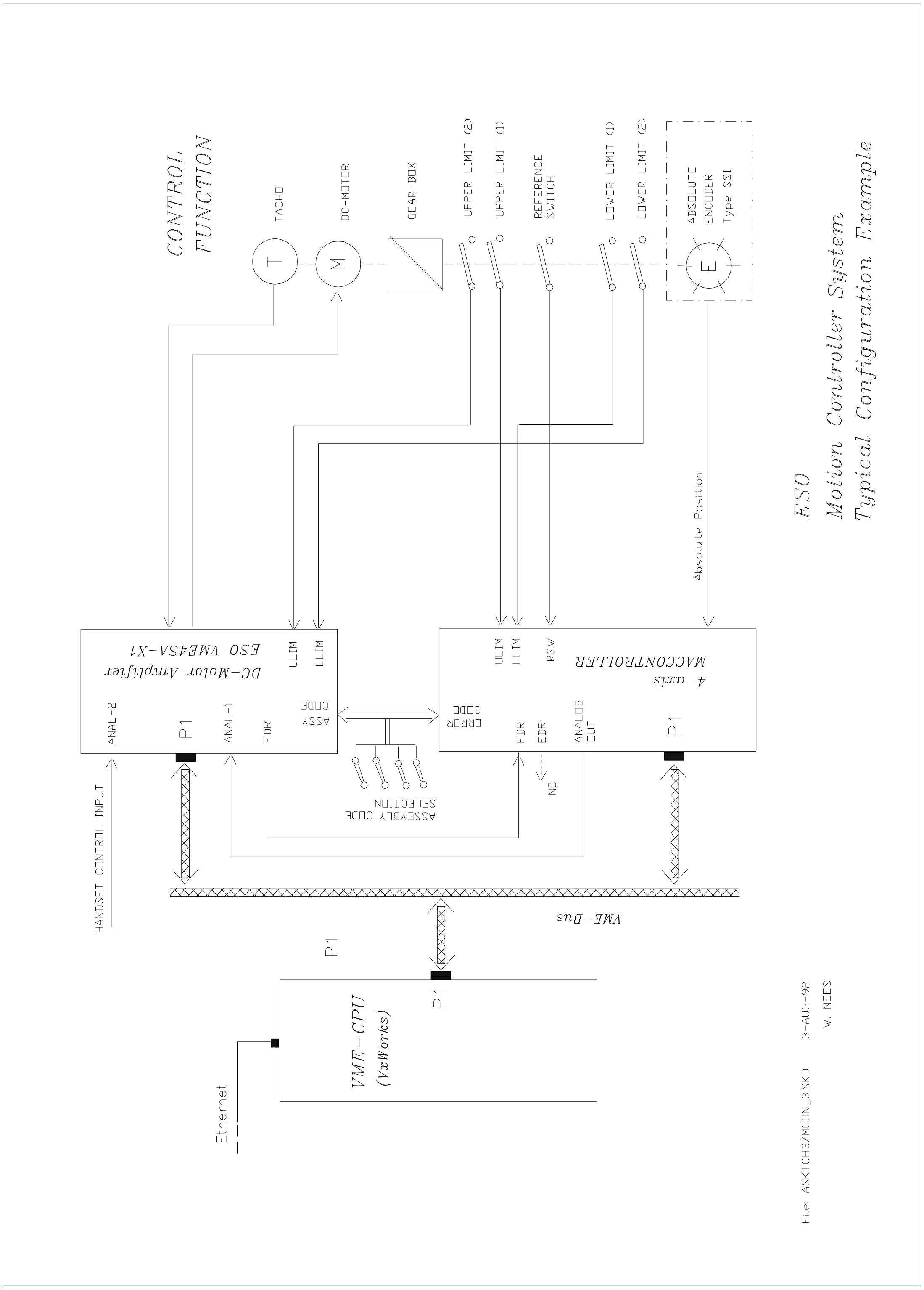
Config23: amplifier, I/O status module, data conversion module, two sets of limit switches, reference switch, arbitrary encoder



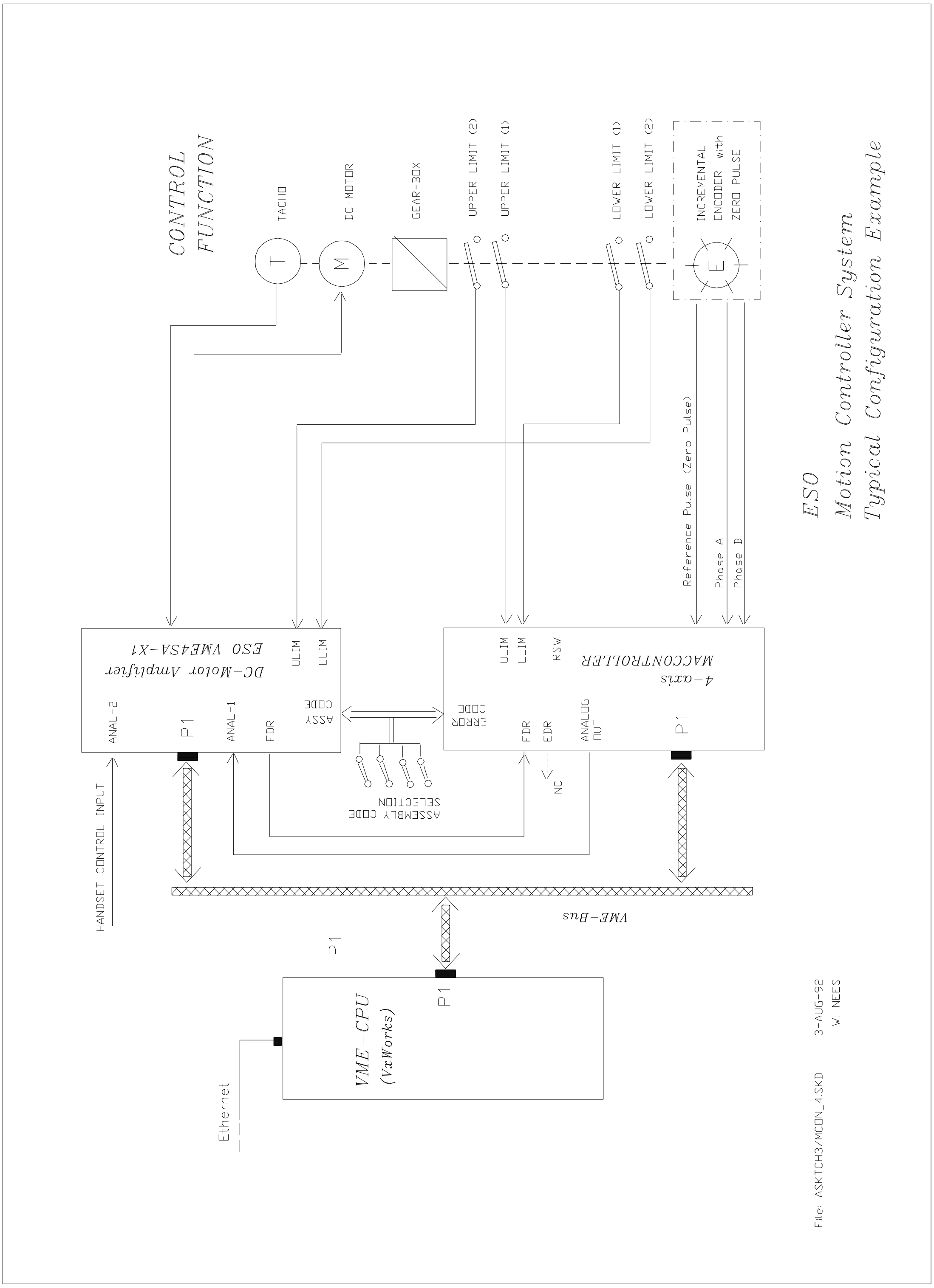
Config1: amplifier, controller, two sets of limit switches, reference switch, incremental encoder with zero pulse



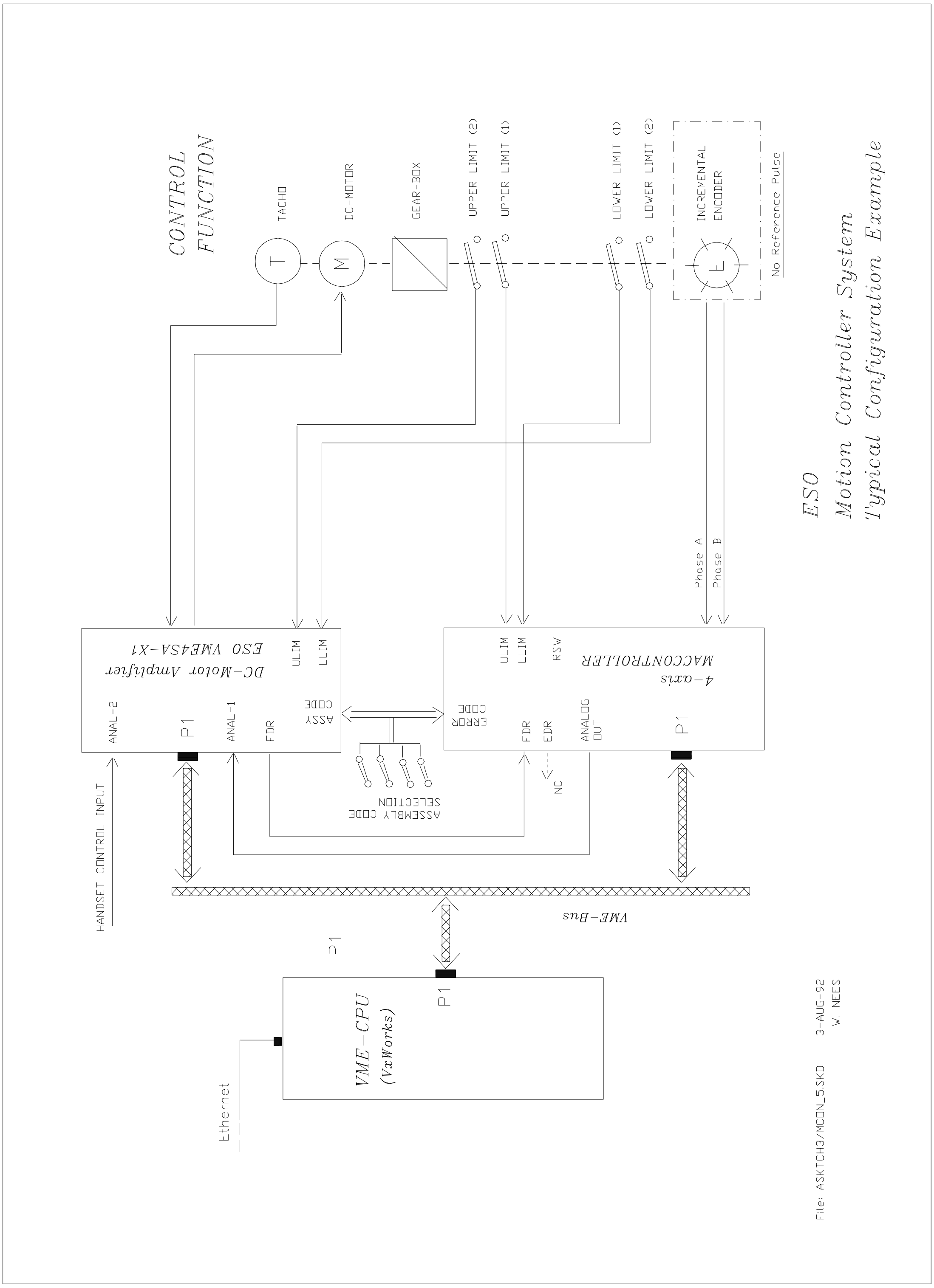
Config2: amplifier, controller, two sets of limit switches, reference switch, incremental encoder without zero pulse



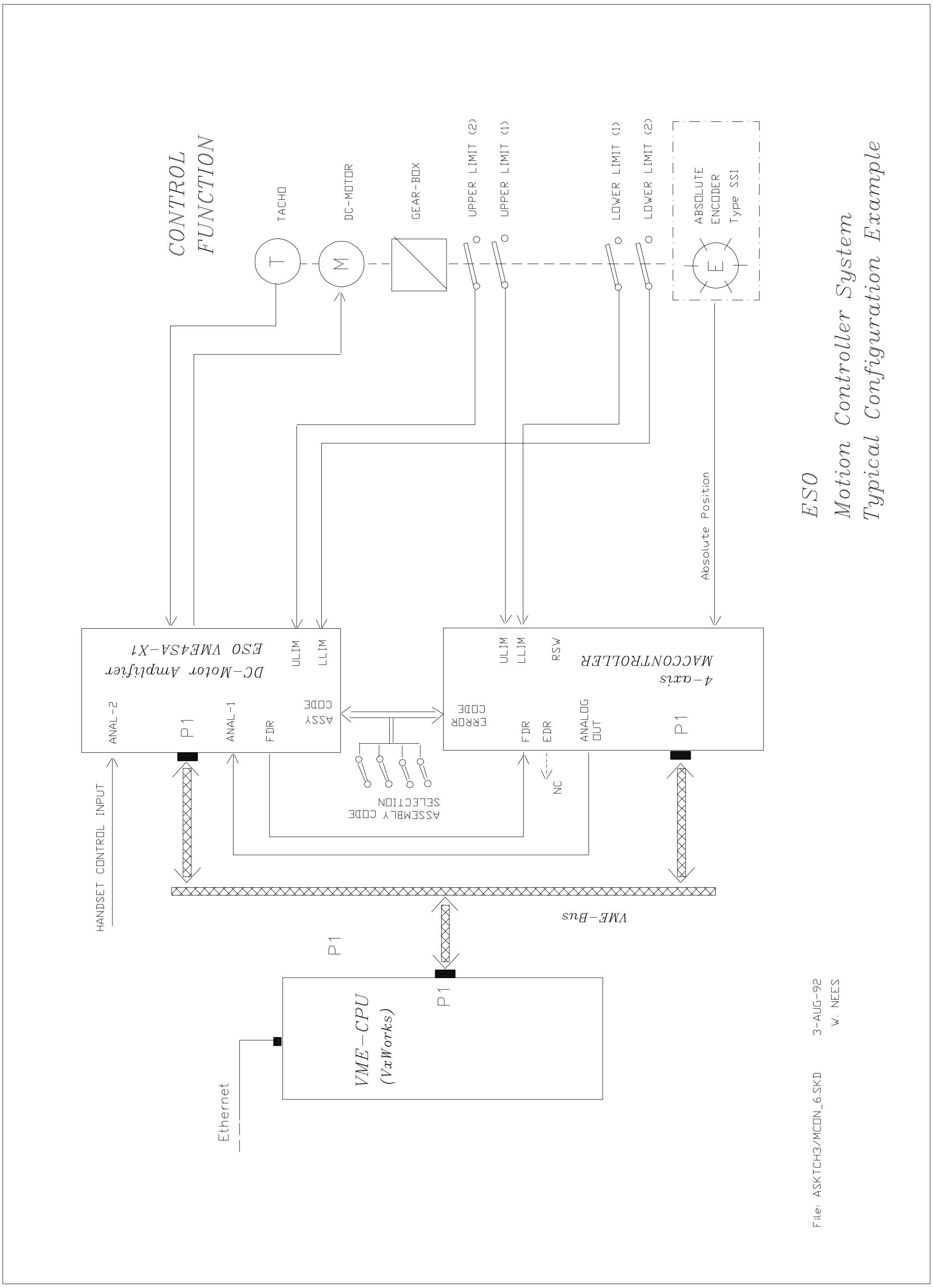
Config3: amplifier, controller, two sets of limit switches, reference switch, absolute encoder



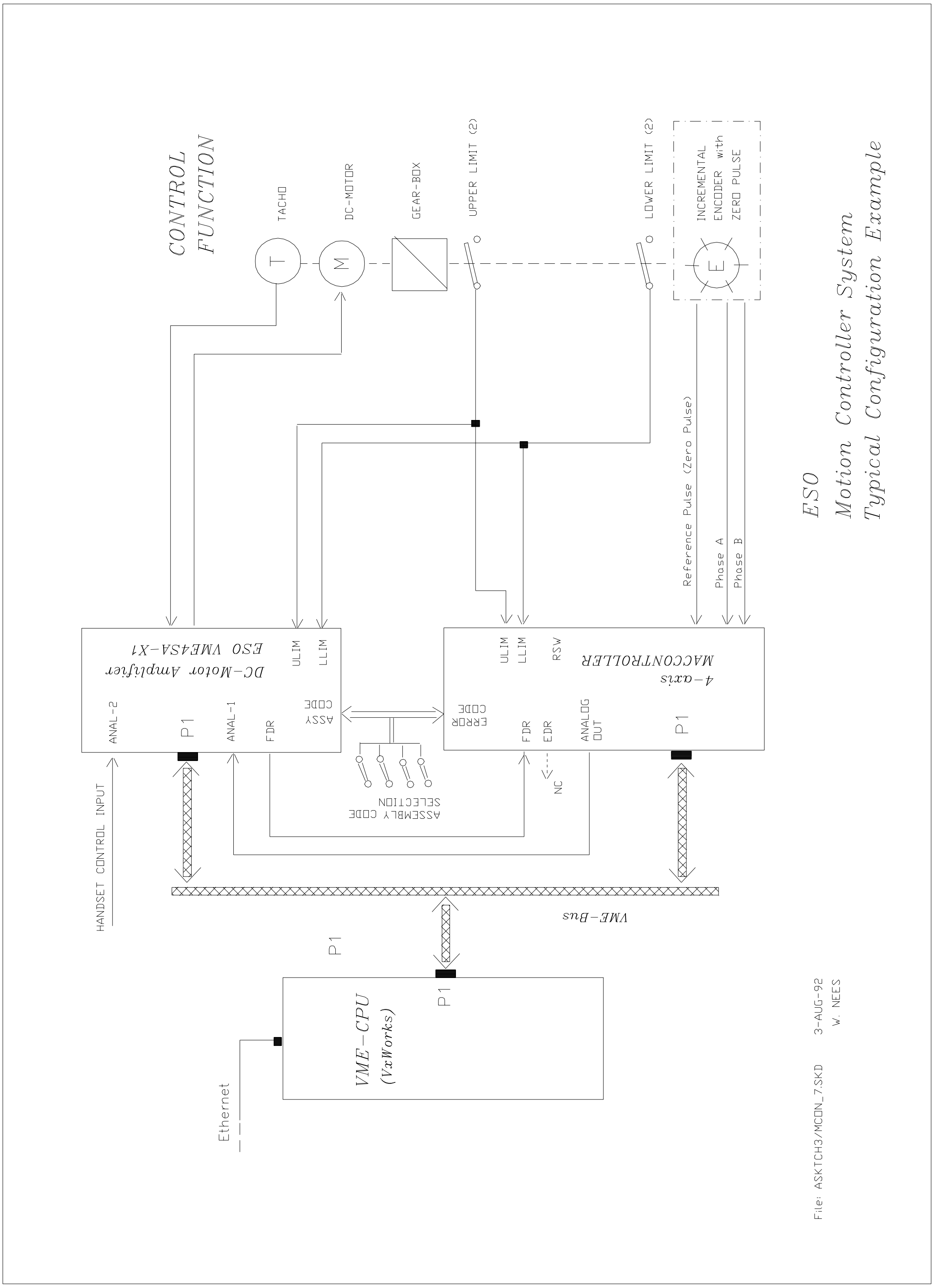
Config4: amplifier, controller, two sets of limit switches, no reference switch, incremental encoder with zero pulse



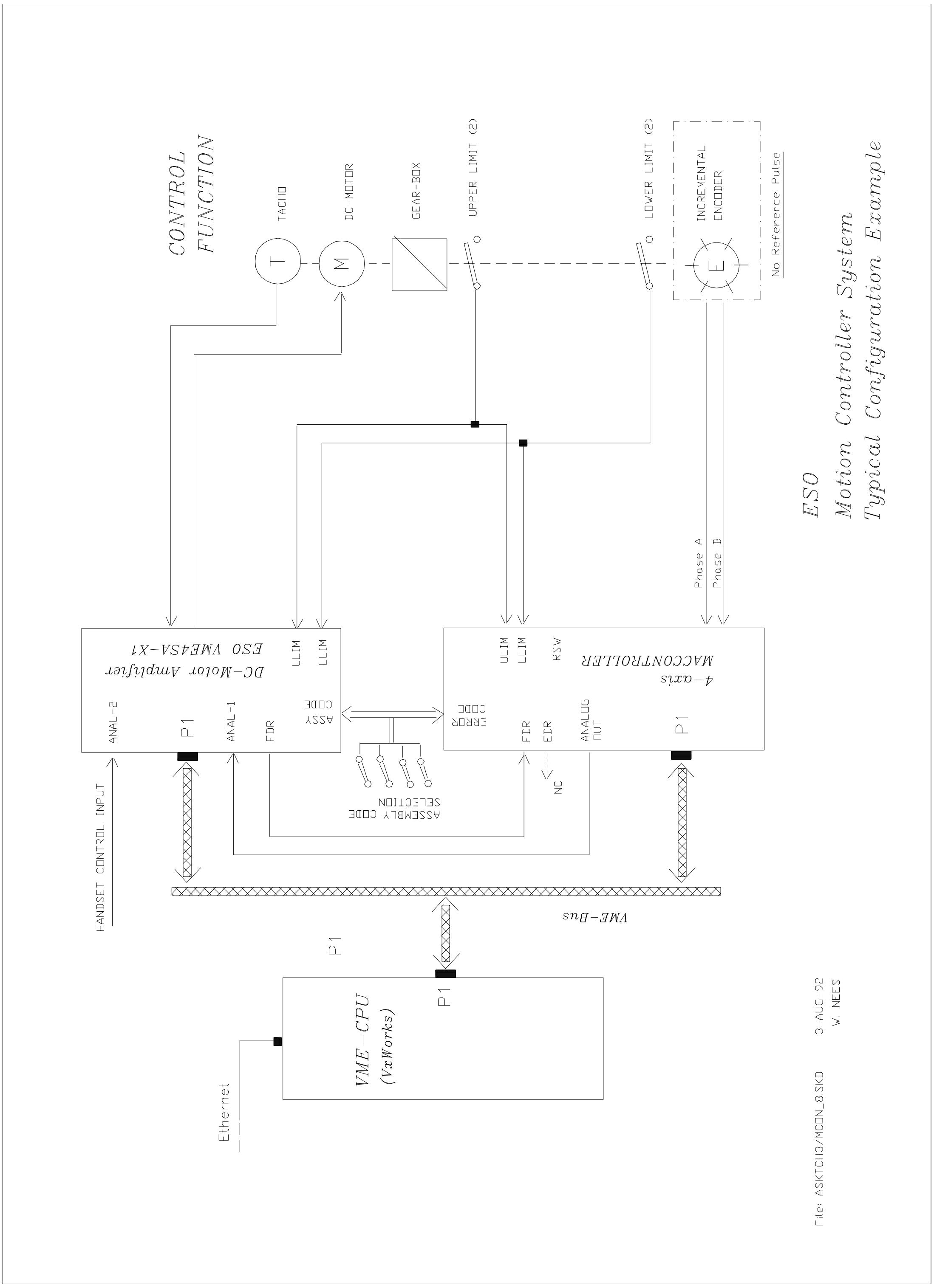
Config5: amplifier, controller, two sets of limit switches, no reference switch, incremental encoder without zero pulse



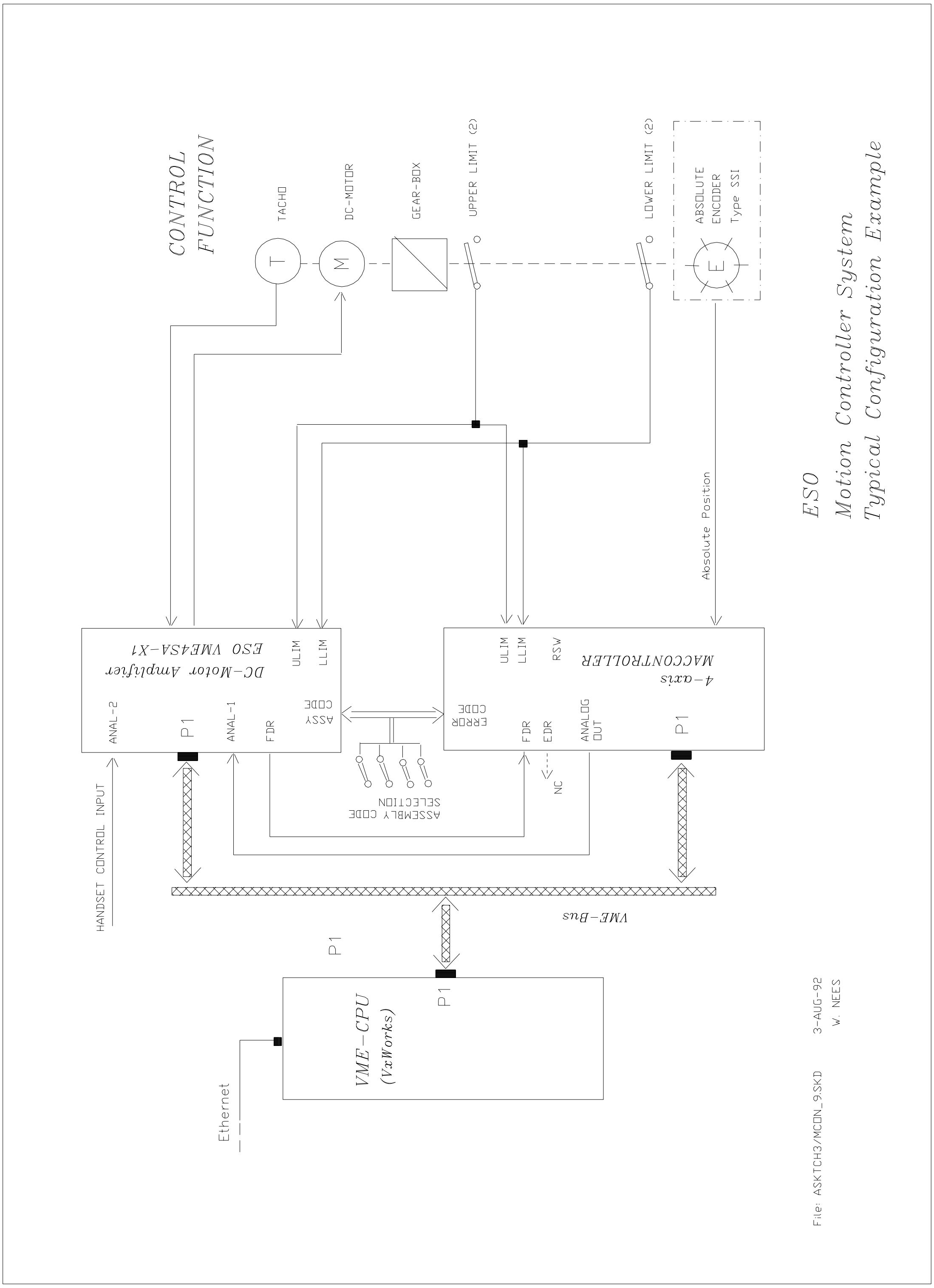
Config6: amplifier, controller, two sets of limit switches, no reference switch, absolute encoder



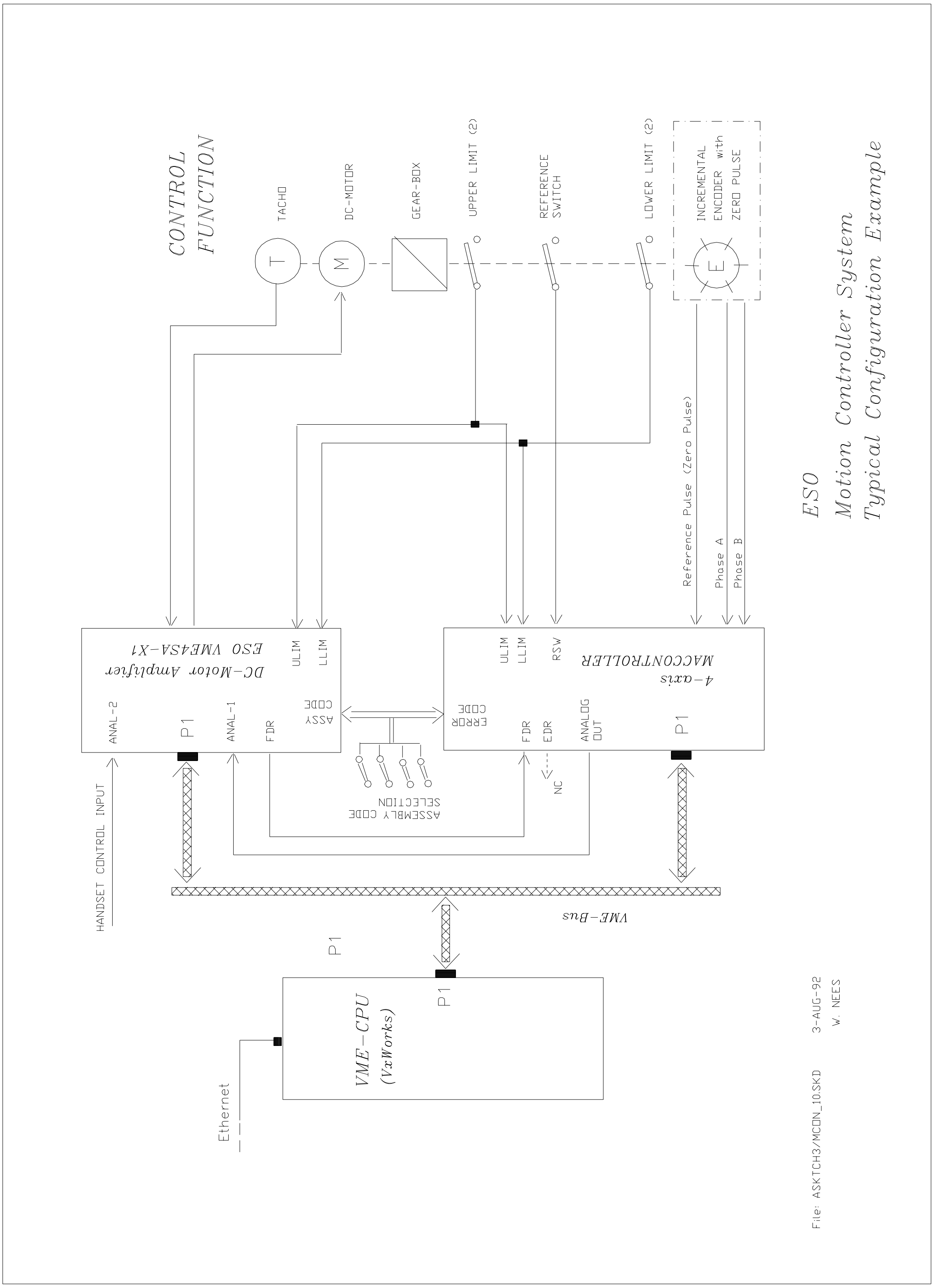
Config7: amplifier, controller, one set of limit switches, no reference switch, incremental encoder with zero pulse



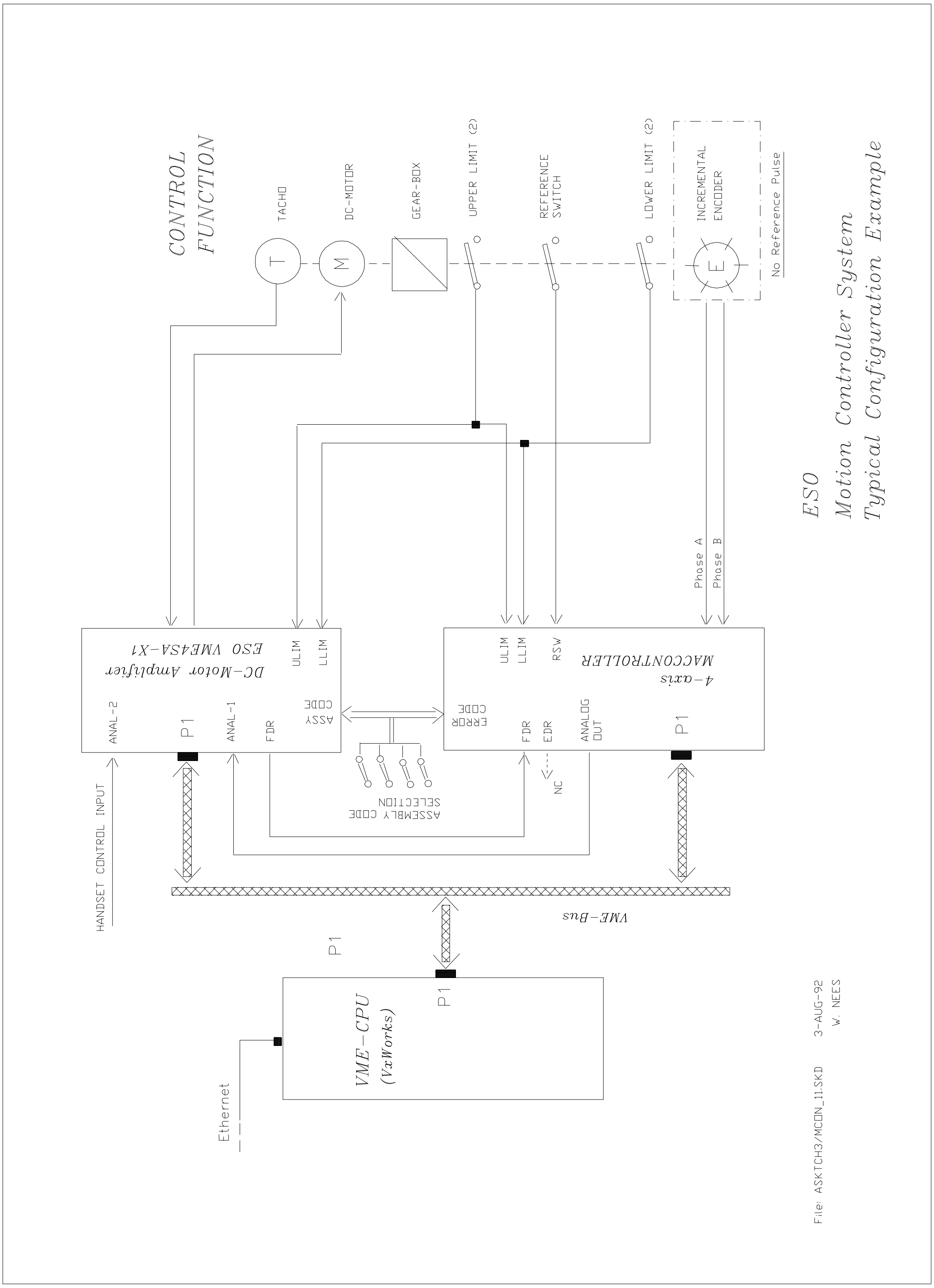
Config8: amplifier, controller, one set of limit switches, no reference switch, incremental encoder without zero pulse



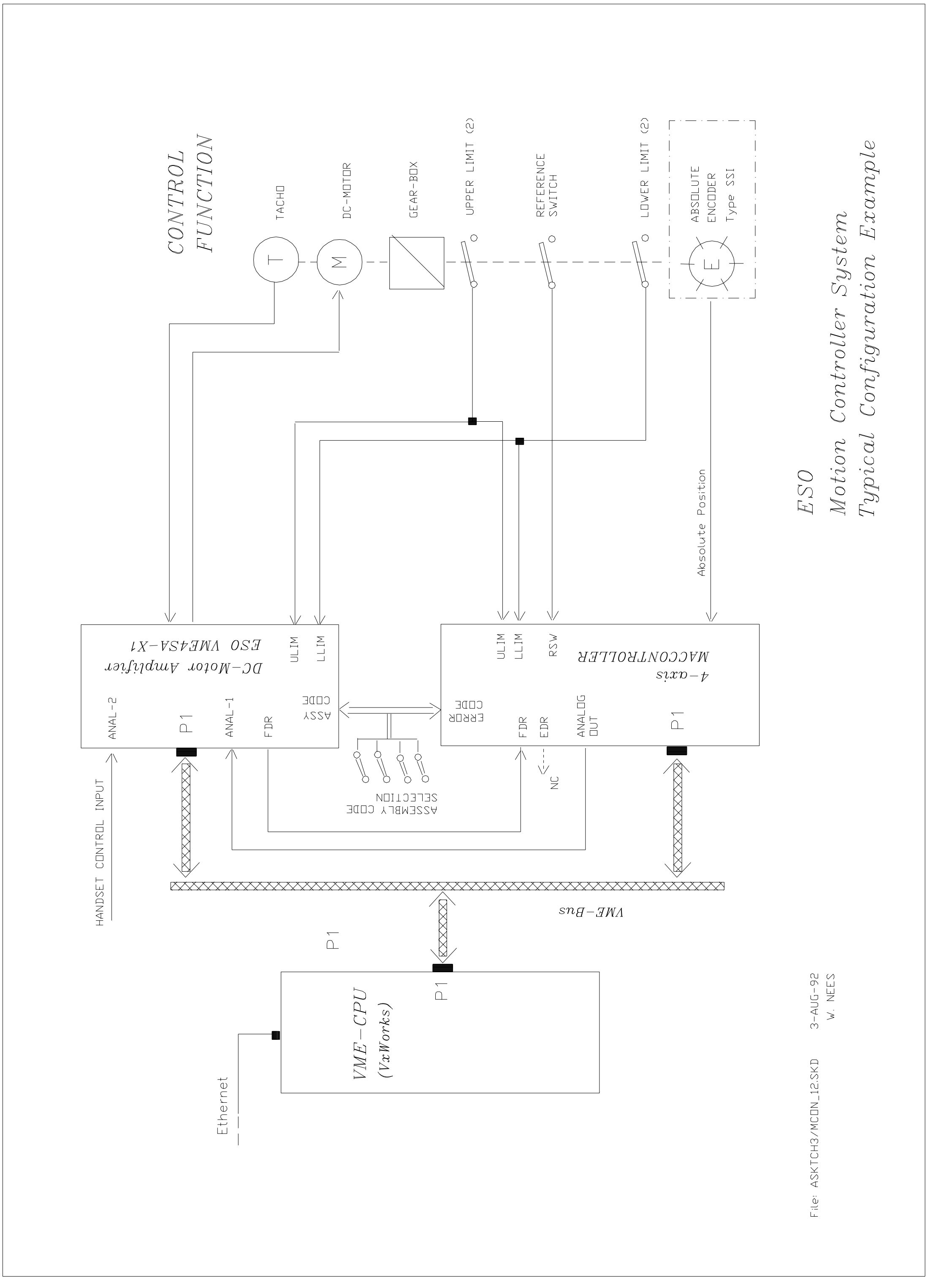
Config9: amplifier, controller, one set of limit switches, no reference switch, absolute encoder



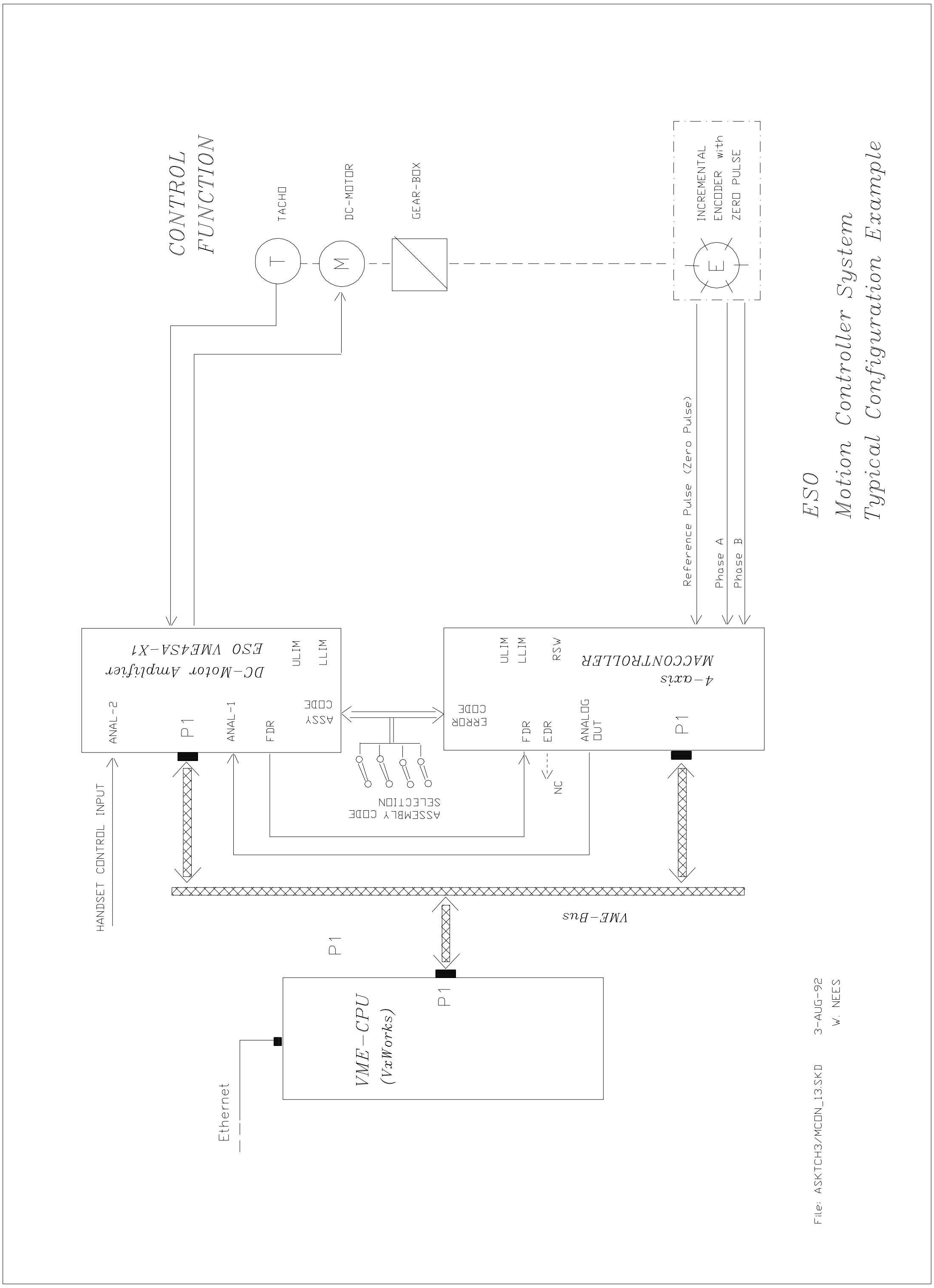
Config10: amplifier, controller, one set of limit switches, reference switch, incremental encoder with zero pulse



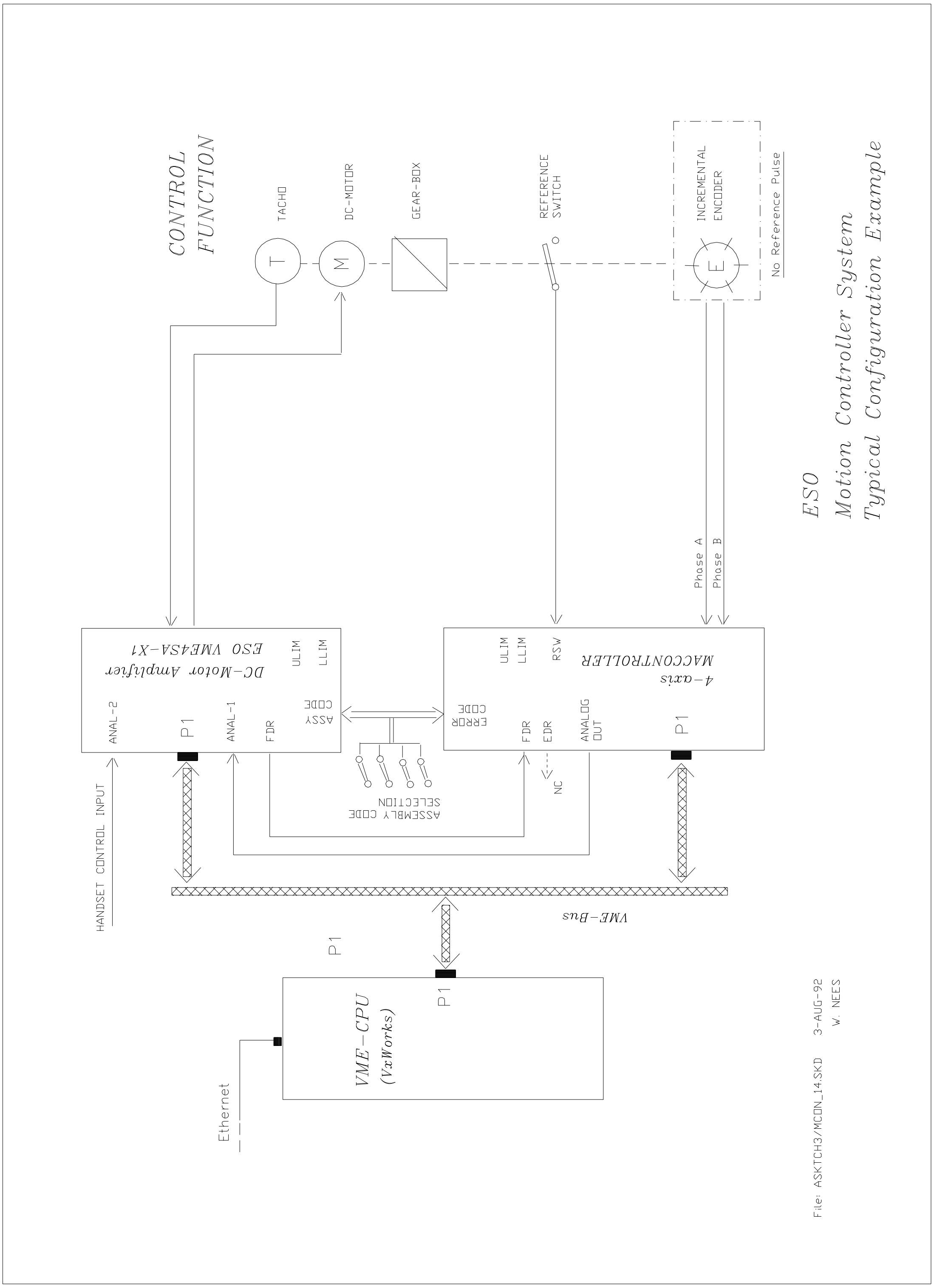
Config11: amplifier, controller, one set of limit switches, reference switch, incremental encoder without zero pulse



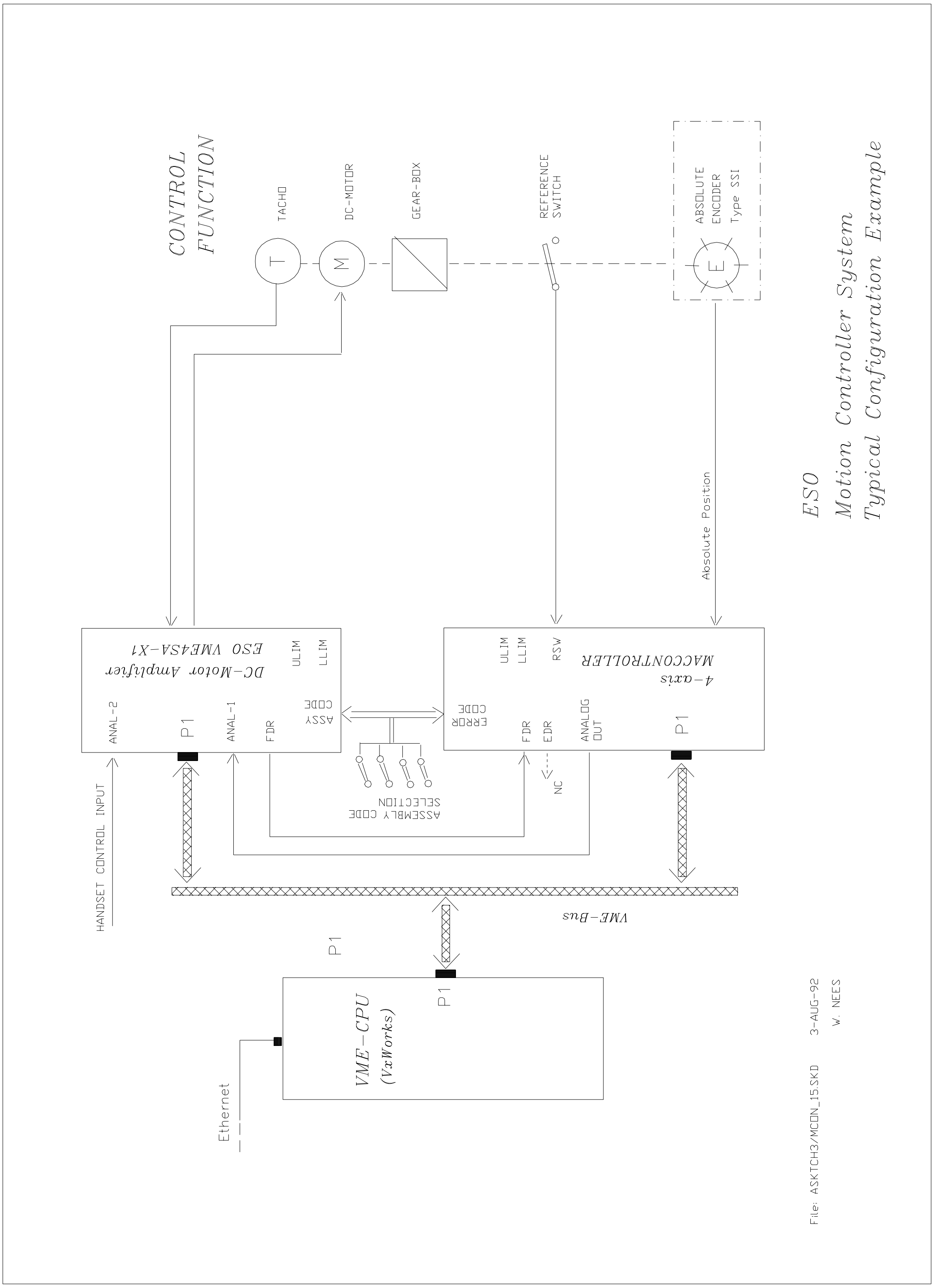
Config12: amplifier, controller, one set of limit switches, reference switch, absolute encoder



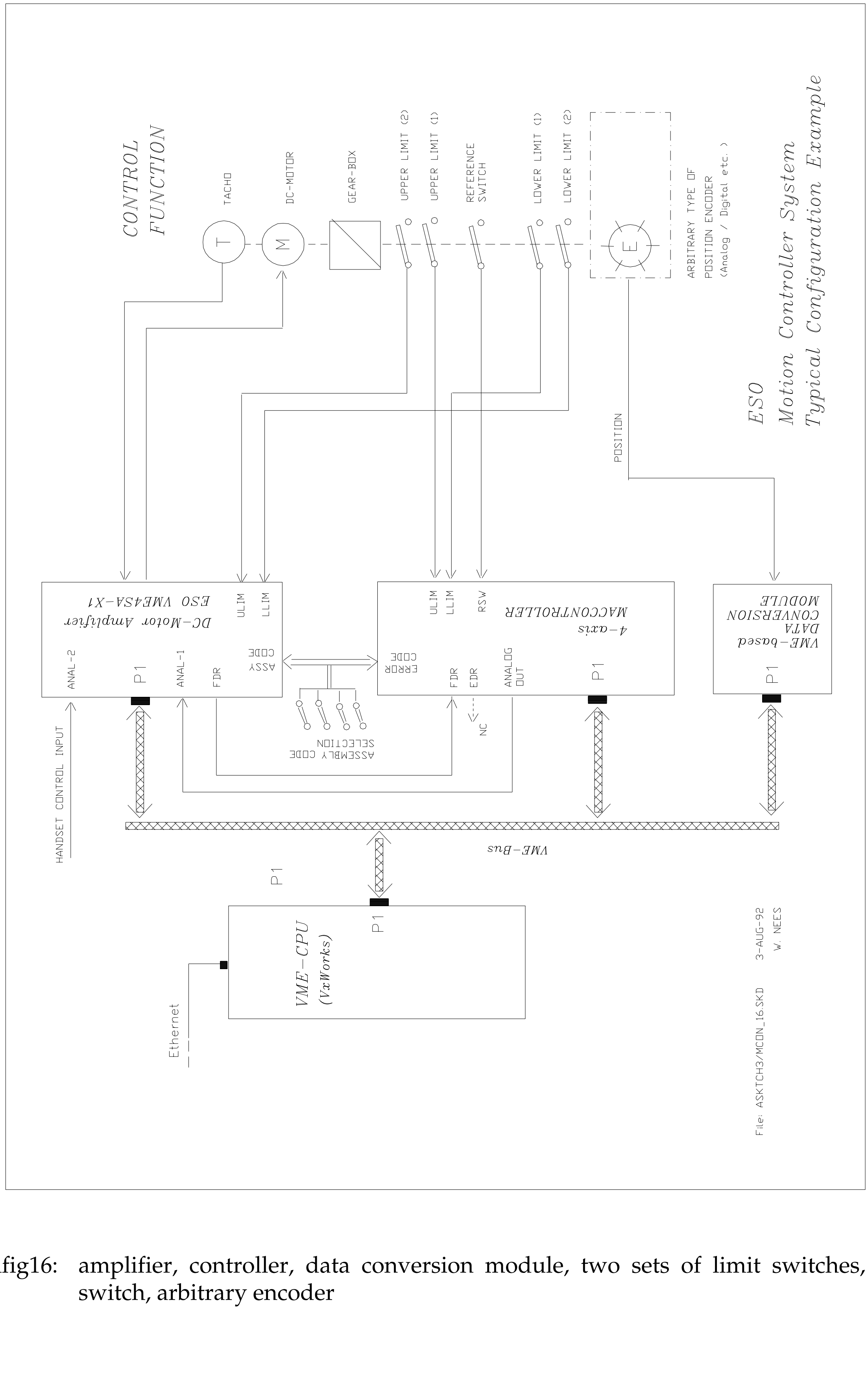
Config13: amplifier, controller, no limit switches, no reference switch, incremental encoder with zero pulse



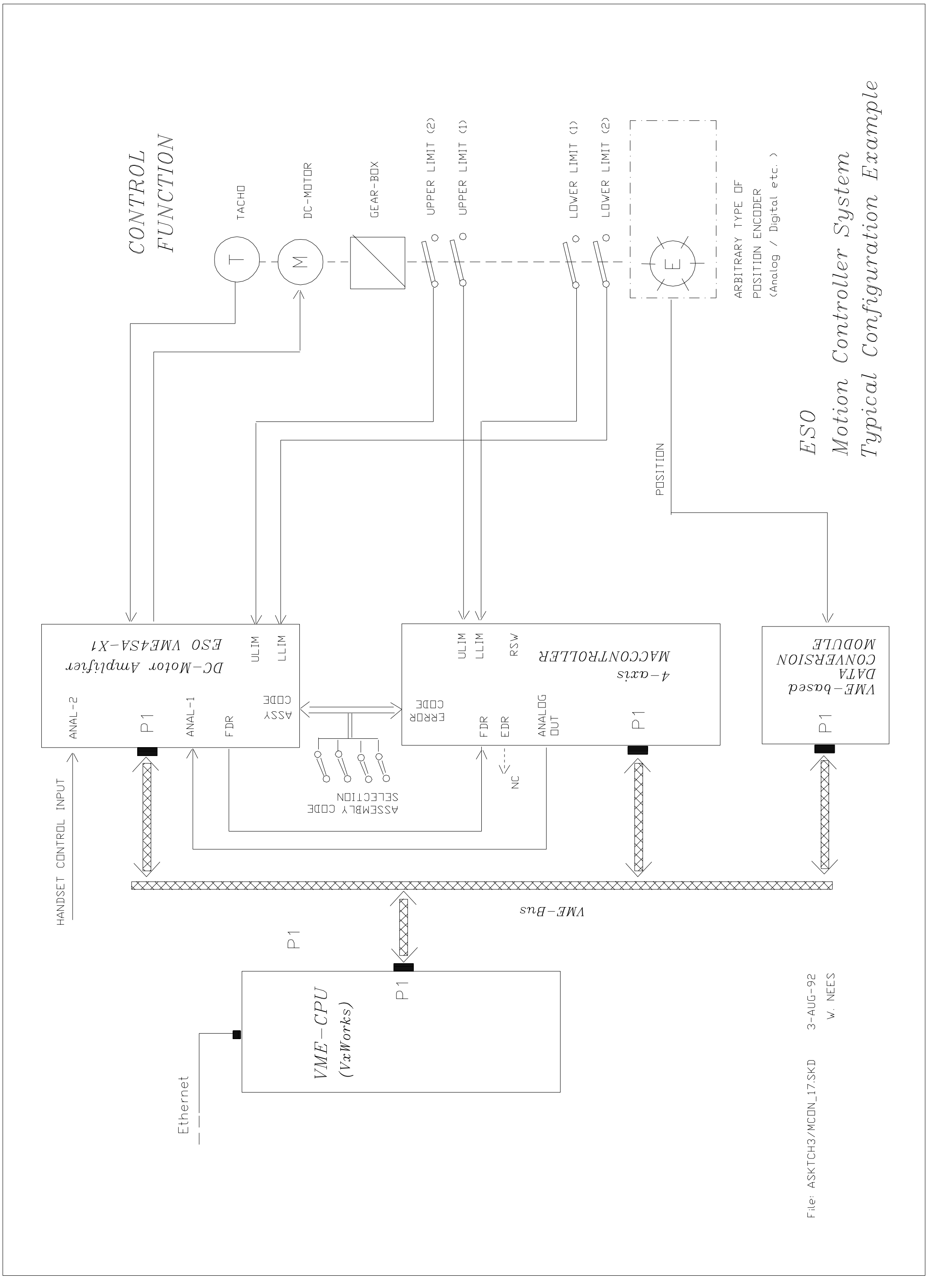
Config14: amplifier, controller, no limit switches, reference switch, incremental encoder without zero pulse



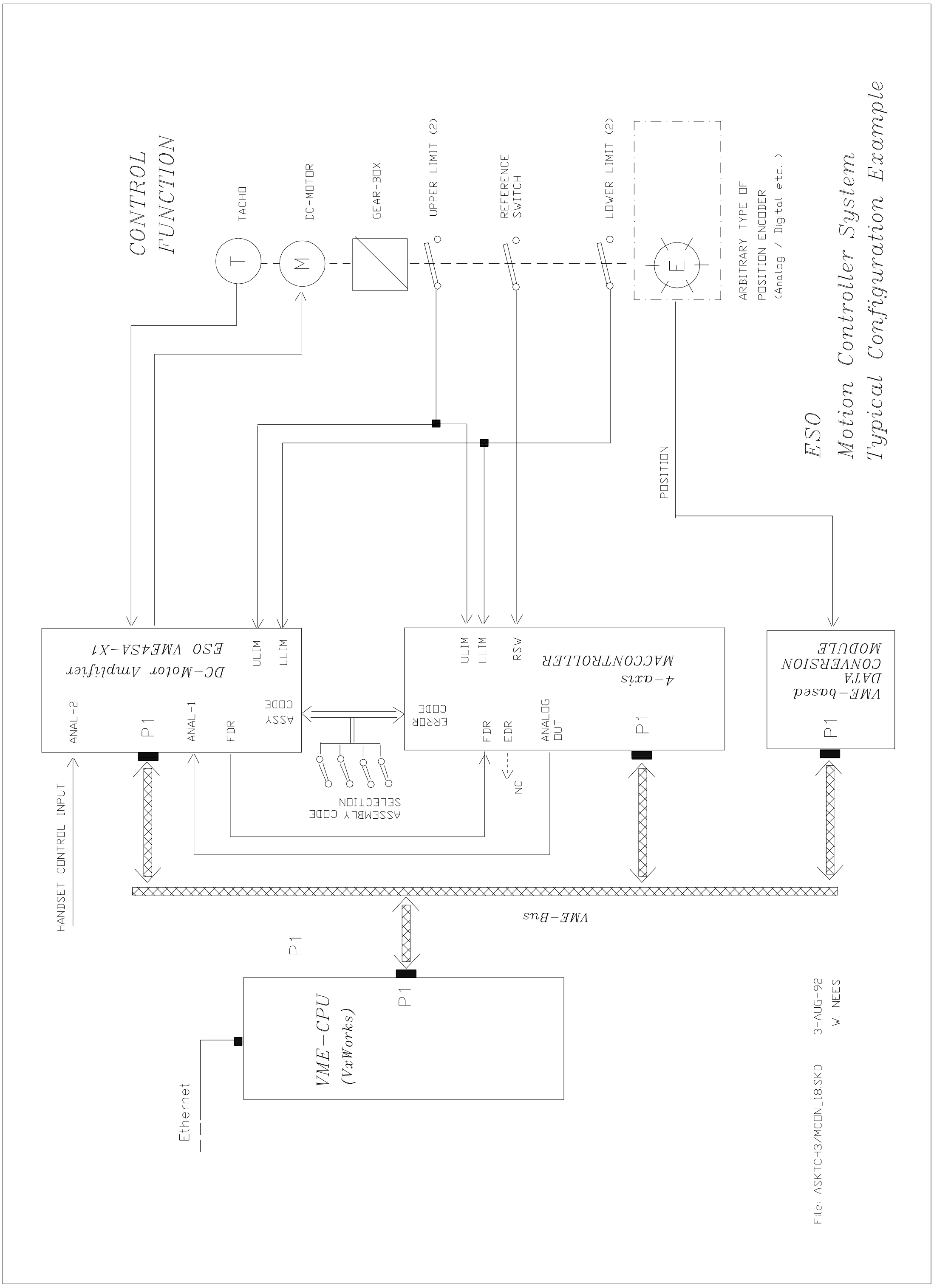
Config15: amplifier, controller, no limit switches, reference switch, absolute encoder



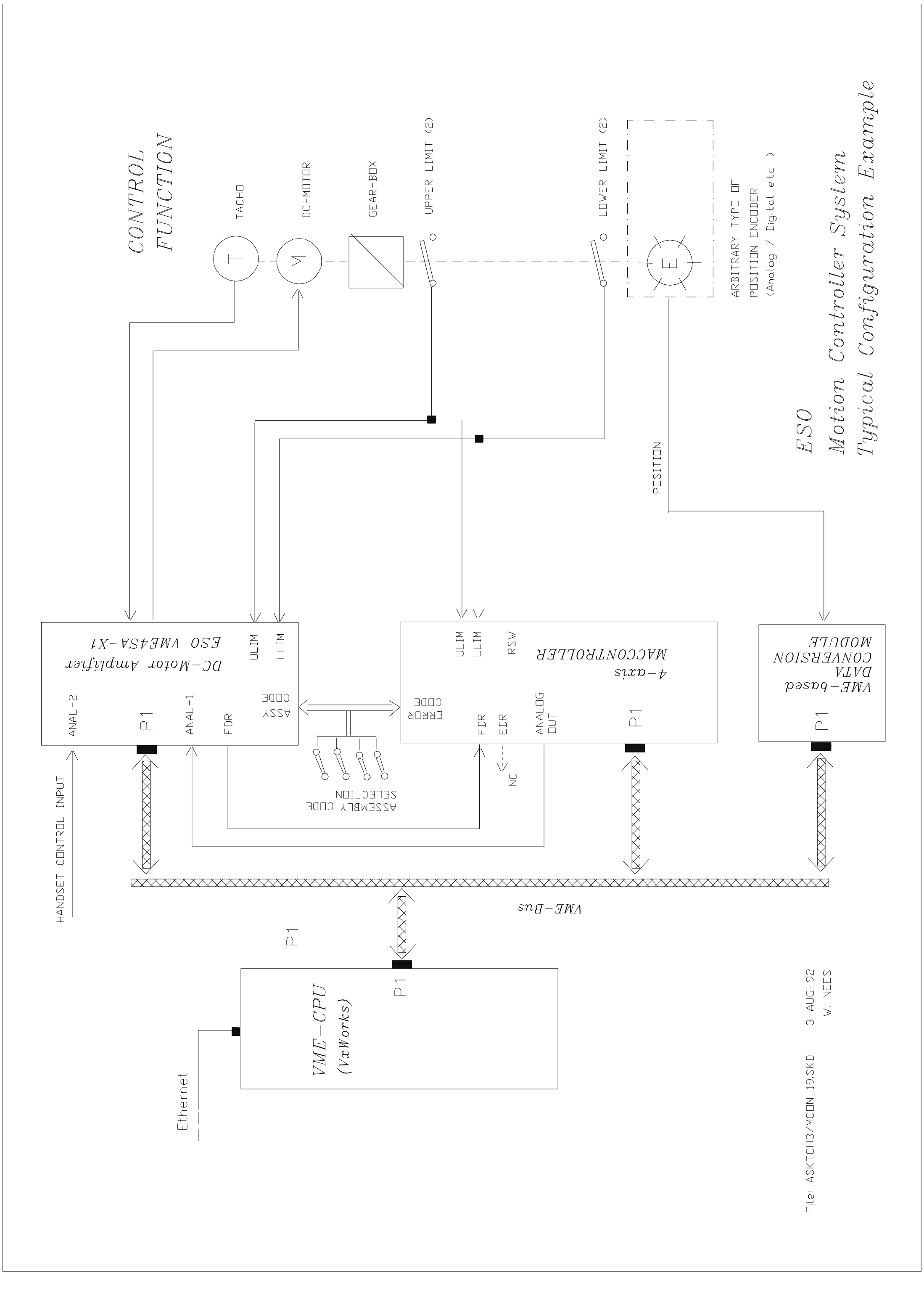
Config16: amplifier, controller, data conversion module, two sets of limit switches, reference switch, arbitrary encoder



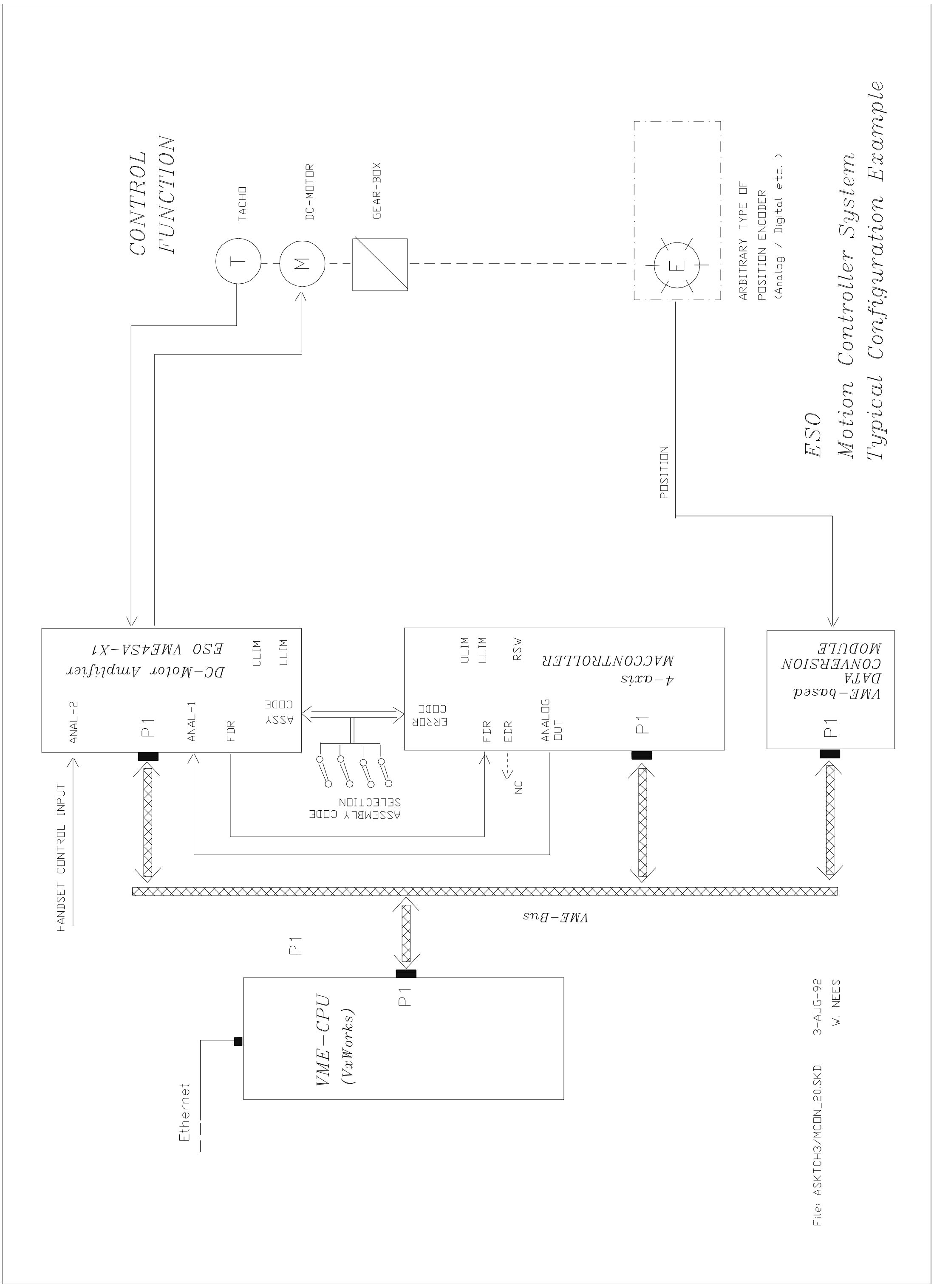
Config17: amplifier, controller, data conversion module, two sets of limit switches, no reference switch, arbitrary encoder



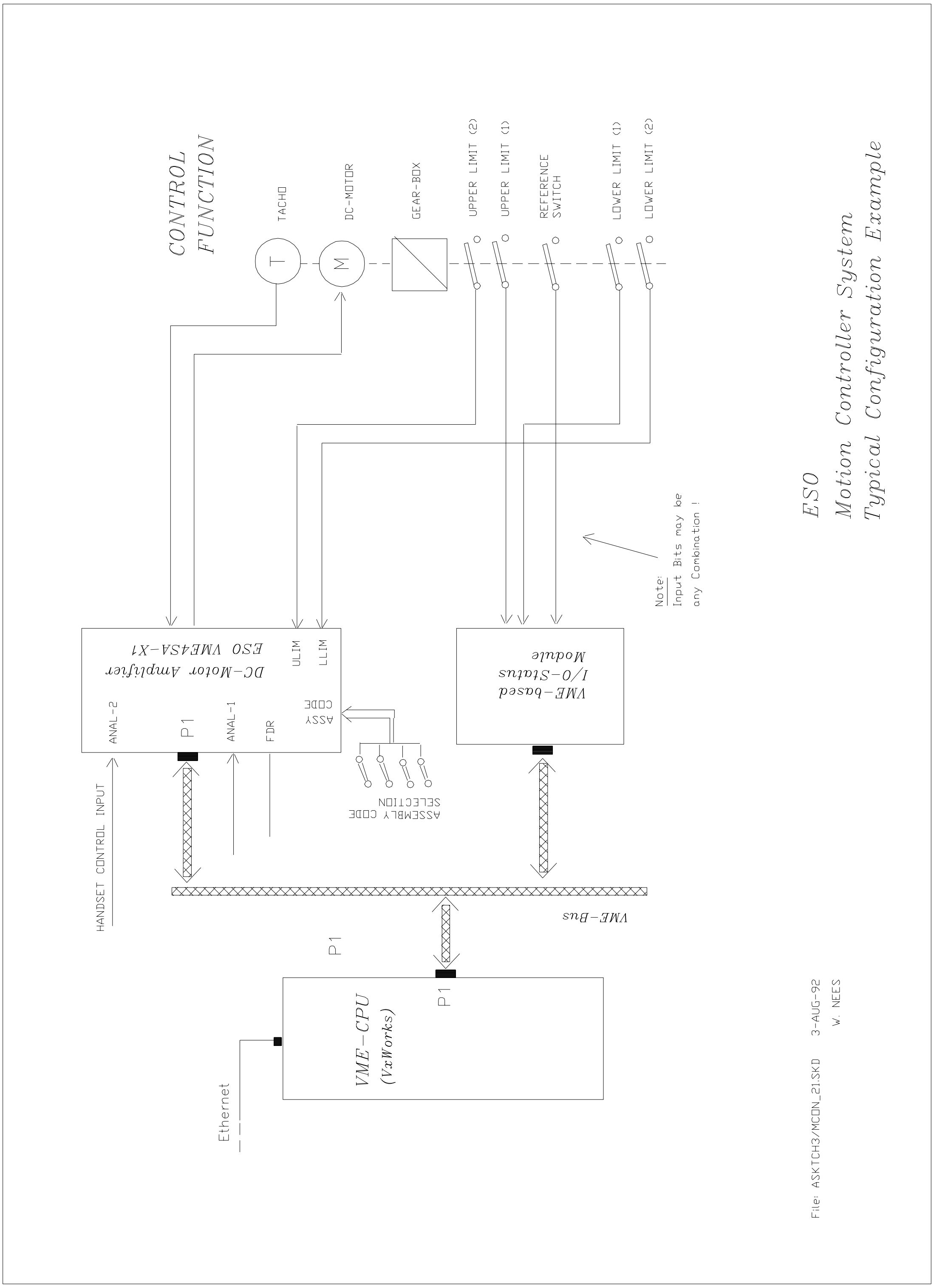
Config18: amplifier, controller, data conversion module, one set of limit switches, reference switch, arbitrary encoder



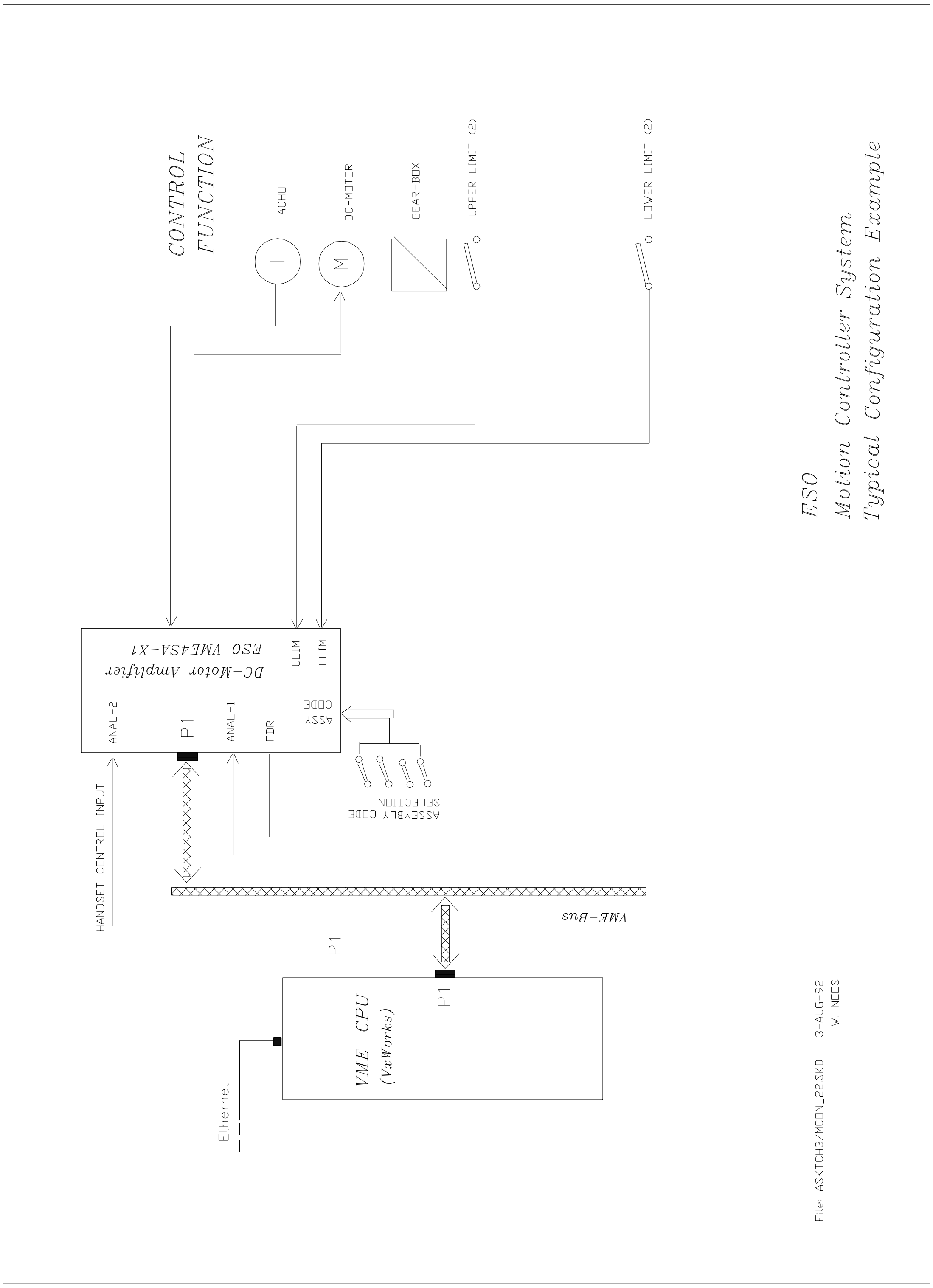
Config19: amplifier, controller, data conversion module, one set of limit switches, no reference switch, arbitrary encoder



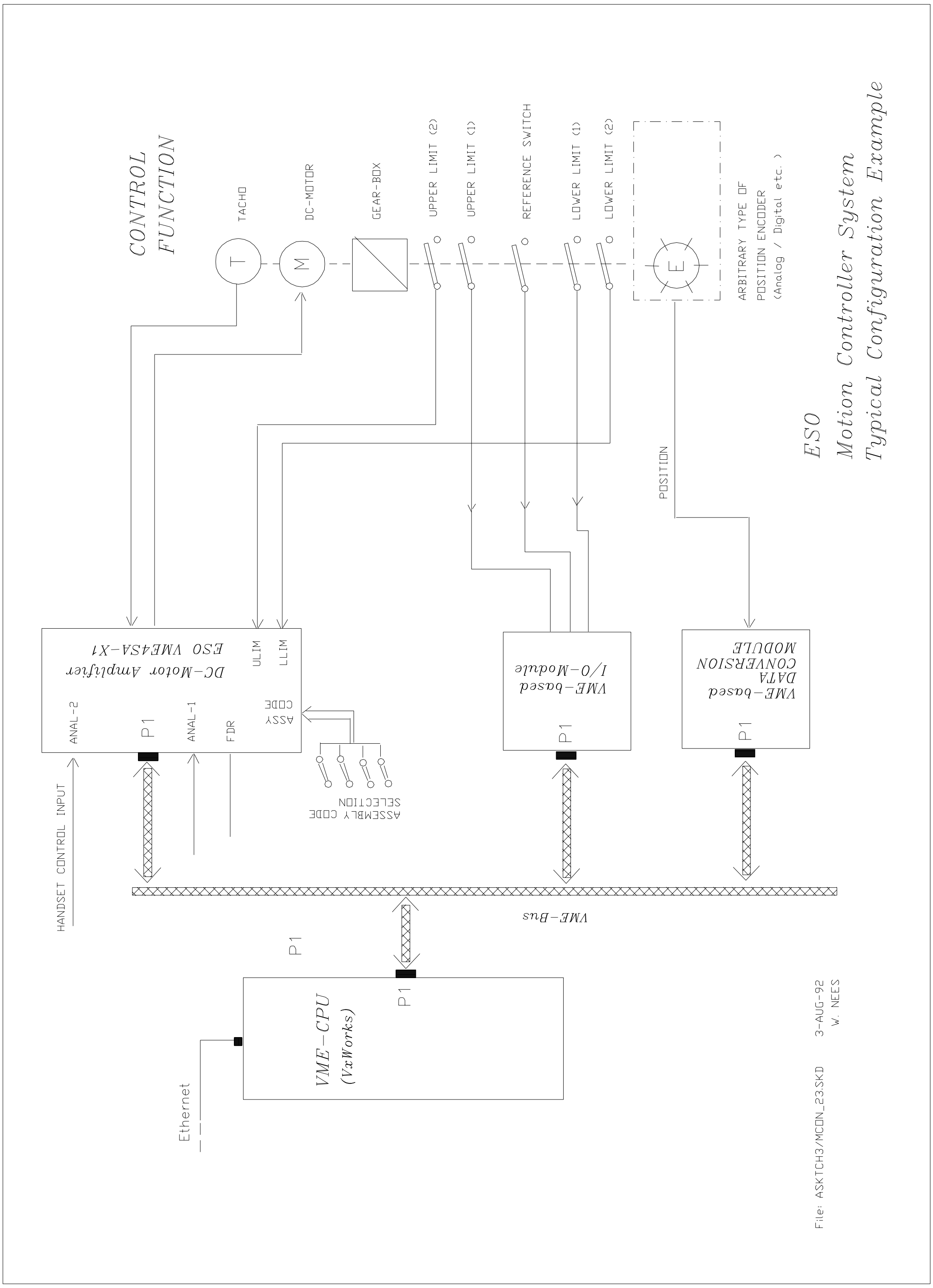
Config20: amplifier, controller, data conversion module, no limit switches, no reference switch, arbitrary encoder



Config21: amplifier, I/O status module, two sets of limit switches, reference switch, no encoder



Config22: amplifier, one set of limit switches, no reference switch, no encoder



Config23: amplifier, I/O status module, data conversion module, two sets of limit switches, reference switch, arbitrary encoder

___oOo___